MASTER EN SCIENCES INFORMATIQUES

Simulation multiagent de situations d'urgence dans le cadre de la RoboCupRescue

Koch, Emmanuel

Award date:
2002

MÉMOIRE

présenté aux

FACULTÉS UNIVERSITAIRES
NOTRE DAME DE LA PAIX

SIMULATION MULTIAGENT DE SITUATIONS D'URGENCE

dans le cadre de la RoboCupRescue

par

Emmanuel Koch

---

pour l'obtention du grade de

MAÎTRE EN INFORMATIQUE

Soutenu en septembre 2002

P.Y. Schobbens ..... Promoteur
B. Chaib-draa ..... Directeur de stage

Table des matières

Remerciements

Je tiens à remercier très chaleureusement le Professeur Pierre-Yves Schobbens pour son soutien et pour m'avoir appuyé dans le choix de mon mémoire, ainsi que le Professeur Brahim Chaib-draa pour ses nombreux conseils et encouragements.

Je souhaite également remercier Morimoto Takeshi pour l'aide importante qu'il m'a apporté dans la réalisation de mon stage, ainsi que Christiane Petit pour ses commentaires, suggestions et pour le soutien qu'elle m'a apporté tout au long de mon stage.

Enfin, je remercie tous les membres de ma famille ainsi que Roxane Jungbluth pour leur soutien de tous les instants. J'adresse ici un remerciement particulier à mon père, Raymond Koch, sans qui ma passion pour l'informatique et en particulier pour l'intelligence artificielle ne serait probablement pas ce qu'elle est aujourd'hui...

Chapitre 1

Introduction

Tantôt créatrice, tantôt destructrice, la nature gouverne depuis des millénaires les formes de vie existant sur terre. Apparition de nouvelles espèces, disparition d'autres, périodes d'abondance, grandes famines, elle dispose de nombreux pouvoirs et ses desseins paraissent bien souvent impénétrables. Un de ses produits mérite d'ailleurs une attention toute particulière : il s'agit de l'Homme. Aurait-elle créé une espèce capable de déjouer, sinon d'entraver ses projets ? Rien n'est moins sûr. Car même si l'Homme, avec tout son arsenal technologique, est capable de s'exterminer lui-même ainsi que la majorité des espèces vivantes sur la planète, il est à peine capable de détecter et de constater, impuissant, l'arrivée d'un violent tremblement de terre, d'un gigantesque ras-de-marée, d'un puissant cyclone, ou peut-être même d'un réchauffement tragique de la planète. Cependant, même si l'Homme ne se bat pas d'égal à égal contre les sursauts destructeurs de la nature, il s'organise et gagne un peu de terrain chaque jour.

Voilà, rapidement esquissé, le contexte dans lequel s'inscrit le sujet dont nous allons traiter dans cet ouvrage. En effet, nous allons vous présenter le travail effectué par des centaines de chercheurs, désireux de nous permettre de réagir le plus efficacement possible lorsque de tels événements catastrophiques surviennent. Il s'agit plus exactement d'un travail d'étude des désastres, sur base de la simulation de leurs conséquences en fonction de notre mode d'organisation et de réaction.

Vous découvrirez ainsi, au travers de ce mémoire, un domaine assez récent de la science informatique, inscrit dans la branche de l'intelligence artificielle, à savoir : «les systèmes multiagents» (Chapitre 2). Ce genre nouveau de systèmes informatiques est notamment utilisé par l'association «RoboCup» pour la simulation de divers thèmes particuliers. Vous aurez l'occasion de les survo-

ler (Chapitre 3), après quoi nous entrerons en profondeur dans les simulations de tremblements de terre du pôle «Rescue» de la RoboCup. Le simulateur multiagent vous sera présenté en détail (Chapitre 4), ainsi que sa population d'«agents» (Chapitre 5). Vous pourrez alors apprécier le travail de simulation multiagent que nous avons réalisé sur base de règles stratégiques originales concernant la gestion d'équipes de secours (Chapitre 6). Nous apporterons aussi quelques critiques et propositions d'évolutions futures concernant les imperfections que vous auriez pu déceler au cours de votre découverte du simulateur (Chapitre 7). Enfin, vous pourrez approcher un genre de sinistres peu connus dans nos régions, au travers d'une approche d'adaptation possible du simulateur de la RoboCupRescue aux tempêtes de verglas (Chapitre 8).

Chapitre 2

Les systèmes multiagents

Le thème des «systèmes multiagents[1]» est à l'intersection de plusieurs domaines, dont celui de l'intelligence artificielle, des systèmes informatiques distribués et du génie logiciel. C'est une discipline qui s'intéresse aux systèmes composés d'entités autonomes appelées «agents» ainsi qu'à leurs comportements collectifs.

Dans ce chapitre, nous allons définir et expliquer ce qu'est un système multiagent[2]. Pour y parvenir, nous commencerons par décrire l'entité qui le compose : l'agent. Cependant, un agent peut fonctionner de bien des manières, et adopter des comportements très divers. C'est pourquoi nous présenterons les différents types d'agents, ainsi que toute une série de propriétés pouvant leur être attribuées. Une fois défini, nous pourrons enfin décrire les systèmes multiagents eux-même et leurs propriétés.

2.1 L'agent logiciel

Pour permettre une compréhension intuitive des agents, ceux-ci sont élaborés en suivant la métaphore des personnes agissant au sein d'une société. Un agent peut donc être vu comme une sorte d'«être» virtuel intelligent et indépendant, situé dans son propre monde, et capable d'interagir avec son environnement et ses semblables tout comme nous sommes capables de le faire dans le monde réel. Cette image est bien sûr une approximation métaphorique mais nous pensons qu'elle peut donner une bonne intuition de ce

[1] On trouve dans la littérature diverses orthographes du mot *multiagent* (multi-agent, multi-agents, multiagent, multiagents). Nous avons décidé d'utiliser une orthographe semblable au mot *pluridisciplinaire*, sans trait d'union, et dont le pluriel n'est pas systématique.

[2] Une grande partie de ce chapitre est inspirée du cours «agents et systèmes multiagents» [Chaib-draa, 1999].

qu'est un agent.

Commençons par présenter l'élément indispensable systèmes multiagents : l'«agent logiciel[3]». Parmi plusieurs définitions trouvées dans la littérature, retenons celle proposée dans [Wooldridge et al., 1998] :

> *Un agent est un système informatique,* ***situé*** *dans un environnement, et qui* ***agit*** *d'une façon* ***autonome*** *et flexible pour atteindre les objectifs pour lesquels il a été conçu.*

Un agent est donc avant tout un programme informatique, un logiciel. Une première particularité de ce logiciel est d'être **situé** dans un environnement. C'est-à-dire que ce logiciel (appelons-le dorénavant «agent») est situé dans un cadre où s'exécutent d'autres programmes, dont certains sont également des agents. Cet agent est capable d'**agir** sur son environnement à partir des entrées (informations) qu'il en reçoit. Il peut demander des services à d'autres programmes et rendre des services. Entre autres, il peut se comporter comme le client d'un serveur, ou comme un serveur pour d'autres clients, voire d'autres agents. Enfin, il est **autonome**, c'est-à-dire qu'il est capable d'agir sans l'intervention d'un tiers (humain ou agent), et contrôle ses propres actions ainsi que son état interne.

Un système multiagent peut donc être vu comme un système distribué particulier, dont les différentes parties sont les agents. On attribue la particularité de ce système au comportement de ses composantes (les agents), souvent difficile à prévoir. En effet, le comportement d'un agent dépend de nombreux paramètres, tels l'état de son environnement (souvent imprévisible), sa position ou ses promotions hiérarchiques dans le système, la présence d'autres agents dans l'environnement ainsi que les coalitions et les conflits qui peuvent apparaître entre eux, l'historique et l'expérience que l'agent peut avoir acquis, les priorités de ses buts ou de ses sous-buts qu'il désirerait atteindre à des moments particuliers...

Sycara et Wooldridge parlent encore de la **flexibilité** des agents. Cela signifie qu'ils doivent être capables de percevoir leur environnement et d'élaborer une réponse dans les temps requis. De plus, ils doivent faire montre d'un comportement proactif et opportuniste, tout en étant capables de prendre l'initiative au «bon» moment. Enfin, ils doivent être «sociaux», c'est-à-dire

[3]Le terme «logiciel» est utilisé ici pour souligner qu'un agent est un programme d'ordinateur. Cependant, nous prendrons l'habitude à l'avenir d'écrire simplement «agent», car les agents logiciels sont notre principal sujet.

capables d'interagir avec les autres agents (logiciels et humains) quand la situation l'exige, afin d'accomplir leurs tâches ou d'aider ces agents à accomplir les leurs[4].

2.1.1 Types d'agents

On sépare généralement les agents en deux grandes classes : les agents dits «cognitifs» et les agents dits «réactifs».

Les agents **cognitifs** reposent généralement sur des **modèles symboliques** (logiques modales et autres...) et des algorithmes de **planification** (quelle action poser et dans quel ordre). En d'autres termes, ces agents possèdent une représentation du monde dans lequel ils évoluent (base de connaissance symbolique) et sont capables de se construire des plans d'actions pour agir sur leur environnement.

Les agents **réactifs**, quant à eux, n'utilisent pas de représentation symbolique ni de raisonnement. Leur comportement intelligent **émerge** de l'interaction entre divers comportements plus simples. Ils sont vus comme des ensembles de comportements accomplissant des tâches données. Chacun de leurs comportements est une **machine à états finis** qui établit une relation entre une entrée sensorielle et une action en sortie. Le développement de ce type d'agents passe par un processus d'observation et d'adaptation de leurs comportements au sein de leur environnement.

Un exemple d'agent cognitif pourrait être un robot de maintenance d'une centrale nucléaire, chargé d'intervenir rapidement lors de problèmes divers. Il posséderait une carte de son environnement pour s'y déplacer efficacement et devrait être capable de planifier ses interventions en fonction de l'état de cet environnement. Un exemple d'agent réactif serait un robot dont le but est de survivre de manière autonome grâce à une source d'énergie émettant un signal. L'agent n'aurait pas de carte de son environnement et s'y déplacerait grâce à des capteurs l'informant d'éventuels obstacles locaux et de la direction de la source d'énergie. Ses règles comportementales pourraient être : se rapprocher de la source si le niveau de ses batteries est trop bas, sinon se déplacer dans une direction quelconque et changer de direction lorsqu'un obstacle se trouve sur son chemin.

[4] Notons qu'avoir la capacité d'aider d'autres agents n'implique pas forcément que l'entraide soit un des buts de l'agent concerné.

Il n'est toutefois pas toujours aisé de classer les agents dans une catégorie plutôt que dans une autre. En effet, que penser d'un agent qui posséderait un modèle symbolique de son environnement, mais qui n'utiliserait qu'une machine à états finis pour parvenir à ses fins ? En fait, on s'est assez vite rendu compte que les agents réactifs convenaient assez bien pour certains types de problèmes qui demandent généralement des réponses simples et rapides (ex : éviter un obstacle), et moins bien pour d'autres problèmes qui dépendent de la qualité de la réponse (ex : prendre le chemin qui minimise l'utilisation d'énergie). Inversément, les agents cognitifs sont moins bien adaptés pour fournir des réponses rapides. Leurs processus de réflexion sont généralement plus lents, tenant compte par exemple de l'évolution possible de leur environnement ou des conséquences que leurs réponses vont impliquer.

C'est ainsi que des architectures hybrides, combinaisons des deux approches, sont nées [Georgeff and Lansky, 1987]. Elles comprennent plusieurs couches (généralement 3) dont les niveaux inférieurs se chargent des comportements purement réactifs, les niveaux intermédiaires de l'environnement, alors que les niveaux supérieurs gèrent la communication avec les autres agents (les aspects sociaux).

Il existe d'autres types d'architectures, parmi lesquelles la plus connue est sans doute l'architecture dite BDI (*Belief, Desire and Intention*). Cette architecture est bâtie autour du raisonnement tel qu'on pense qu'il se pratique chez les humains. Le raisonnement est alors construit sur la prise en compte d'états mentaux :

- *Belief* : Les croyances (Ce que l'agent connaît de son environnement) ;
- *Desire* : Les désirs (Les états possibles vers lesquels l'agent peut vouloir s'engager) ;
- *Intention* : Les intentions (Les états envers lesquels l'agent s'est engagé, et dans lesquels il a engagé des ressources).

2.1.2 Propriétés des agents

Plusieurs chercheurs ont essayé de classer les agents et les systèmes multiagents à partir d'un ensemble de propriétés. A notre connaissance, il n'existe pas de classification universelle, mais celle présentée dans [Chaib-draa, 1999] nous semble raisonnable.

Les propriétés des agents y sont décomposées en deux sous-groupes, à savoir les propriétés intrinsèques et extrinsèques. Les propriétés intrinsèques

concernent sa durée de vie, son architecture, sa construction, sa mobilité, et finalement son adaptabilité. Dans le sous-groupe des propriétés extrinsèques, nous trouvons son endroit d'exécution, son autonomie, sa sociabilité, sa bienveillance, et finalement ses interactions avec les autres agents, vue sous deux aspects : logistique et sémantique.

Propriétés	**Étendue des valeurs**
Durée de vie	de transitoire à longue
Architecture	de réactif à cognitif
Construction	de déclarative à procédurale
Mobilité	de fixe à itinérante
Adaptabilité	de fixe à ajustée ou autodidacte

FIG. 2.1 – Propriétés intrinsèques des agents

La durée de vie de l'agent représente la durée pendant laquelle l'agent va rester dans le système multiagent. Des agents peuvent être créés uniquement lorsqu'on a besoin d'eux, pour disparaître sitôt leur tâche accomplie. Inversément, un agent peut être nécessaire du début à la fin de l'application. La propriété architecture concerne le type de l'agent, cognitif ou réactif, tel que nous l'avons présenté au point 2.1.1. La construction dépend principalement du type de langage de programmation utilisé, à savoir si l'agent a été programmé à l'aide de règles de production (programmation déclarative), ou à l'aide de procédures (programmation procédurale). La mobilité de l'agent concerne sa capacité à se déplacer d'un système multiagent à l'autre. L'adaptabilité d'un agent représente son pouvoir d'adaptation aux changements de son environnement (si une ressource vient à manquer, par exemple). Est-il préparé à ces changements ? Est-il capable d'expérimenter et d'apprendre par lui-même les comportements qui conviennent le mieux en fonction de la situation ?

Propriétés	Étendue des valeurs
Endroit	de local à distant
Autonomie sociale	de indépendante à contrôlée
Sociabilité	autistique, conscient, responsable
Bienveillance	coopératif, compétitif, antagoniste
Interaction	Logistique : direct ou via des intermédiaires Sémantique : communication déclarative ou procédurale

FIG. 2.2 – Propriétés extrinsèques des agents

L'endroit d'exécution est à mettre en rapport avec la possibilité de pouvoir exécuter les agents à distance du système multiagent. Par exemple, certains systèmes multiagents lancent eux-même leurs agents en local, alors que d'autres acceptent l'entrée d'agents mobiles venant d'endroits délocalisés. L'autonomie sociale d'un agent concerne sa prédisposition à être contrôlé par un agent hiérarchiquement supérieur. Notons qu'il n'existe pas toujours de structure hiérarchique dans les systèmes multiagents. La propriété de sociabilité d'un agent dépend des rapports qu'il entretient avec ses semblables. Il peut être autiste ou conscient, c'est-à-dire considérer les autres agents soit comme de simples composantes de son environnement, soit comme des collègues avec lesquels il peut communiquer et coopérer, voire même entrer en conflit. Sa bienveillance va alors dépendre du type de rapports qu'il pourra avoir avec ses semblables. Enfin, les interactions d'un agent avec ses semblables sont à mettre en rapport avec la manière dont il communique avec eux. L'aspect logistique s'intéresse à l'utilisation d'intermédiaires dans la communication, alors que l'aspect sémantique s'intéresse au contenu de ces communications, plus précisément à la manière dont sont construits les messages que les agents vont s'échanger.

2.2 Les systèmes multiagents

On appelle «système multiagent» un système informatique composé de plusieurs agents logiciels. Ce sont donc aussi des systèmes distribués puisqu'ils sont composés d'ensembles d'agents, ces derniers étant tantôt les clients du système distribué, tantôt les serveurs.

On retrouve en général dans les systèmes multiagents les quatre caractéristiques suivantes :

- le point de vue partiel de chaque agent, conséquence de leur capacité limitée à résoudre des problèmes seuls et à leur connaissance limitée de leur environnement ;
- l'absence de contrôle global sur le système, puisqu'il est composé d'une multitude d'entités autonomes ;
- la décentralisation des données, découlant du fait que chaque agent possède une partie des informations du système ;
- le calcul asynchrone, autre conséquence de l'autonomie des agents. En effet, chacun de ces agents traite ses informations de sa propre manière et à son propre rythme.

Les systèmes multiagents bénéficient des avantages propres à la résolution distribuée et concurrente des problèmes (comme la modularité, le parallélisme et la redondance...). De plus, ils héritent des avantages de l'intelligence artificielle (tels que le traitement symbolique, la réutilisation...). Enfin, ils offrent surtout l'avantage de faire intervenir des schémas d'interactions sophistiqués, tels que la coopération (travailler ensemble pour atteindre un même but), la coordination (s'organiser pour éviter les interférences et exploiter les synergies), ou encore la négociation (parvenir à un accord acceptable pour toutes les parties concernées).

Les systèmes multiagents, de par les possibilités d'analyse théorique et expérimentale des mécanismes sous-jacents aux interactions de plusieurs entités autonomes, suscitent l'intérêt des sciences cognitives, sociales et naturelles. Ils constituent également une nouvelle approche de l'analyse, de la conception et de la mise en oeuvre de systèmes informatiques complexes. En effet, de nombreux chercheurs pensent qu'une vision basée sur les agents offre de puissants répertoires d'outils, de techniques et de métaphores qui devraient aboutir à un nouveau paradigme similaire au paradigme objet : le paradigme agent [Chaib-draa, 1999, Jennings, 2000]. La programmation orientée agent aurait, selon eux, le potentiel d'améliorer considérablement les systèmes logiciels. L'avantage viendrait premièrement du degré d'autonomie des agents, car plutôt que d'invoquer une méthode, un agent recevra une requête et décidera de son propre gré s'il doit poser ou non une action. Deuxièmement, il proviendrait du caractère flexible des agents (réactif, pro-actif et social), et troisièmement du fait qu'on considère chaque agent comme une source de contrôle au sein du système alors qu'un système orienté objet n'en possèdera qu'une seule.

2.2.1 Propriétés des systèmes multiagents

Toujours d'après la classification trouvée dans [Chaib-draa, 1999], nous pouvons, indépendamment des propriétés de chaque agent du système, donner au système multiagent des attributs, qui sont : l'unicité, la granulosité, la structure de contrôle et l'autonomie.

Propriétés	Étendue des valeurs
Unicité	de homogène à hétérogène
Granulosité	de fine à grosse
Structure de contrôle	de hiérarchique à démocratique
Autonomie d'interface	communication : vocabulaire spécifique, langage, protocole capacité : buts, croyances, procédures, ontologies
Autonomie d'exécution	de indépendant à contrôlé

FIG. 2.3 – Propriétés des systèmes multiagents

La propriété d'unicité d'un système multiagent sera dite homogène si tous les agents possèdent les mêmes propriétés et se ressemblent, sinon il sera dit hétérogène. La granulosité représente le degré de détail des informations manipulées par ce système (la modélisation d'une maison par un carré sera de granulosité moins fine que par la liste de ses arrêtes en trois dimensions). La structure de contrôle représente le fonctionnement sociétal des agents. En effet, dans le cas d'une structure hiérarchique, les agents doivent obéir à certaines règles qui n'existent pas dans la structure démocratique (une file d'attente avec ou sans priorités, par exemple). L'autonomie d'interface concerne la communication et les capacités nécessaires à l'utilisation de l'interface par un agent pour entrer dans le système. Enfin, l'autonomie d'exécution des agents représente le niveau de contrôle du système sur la liberté de comportement des agents.

On peut encore classer les systèmes multiagents en fonction de leur **cadre d'exécution** (*framework*) et en fonction de leur interaction avec l'**environnement** :

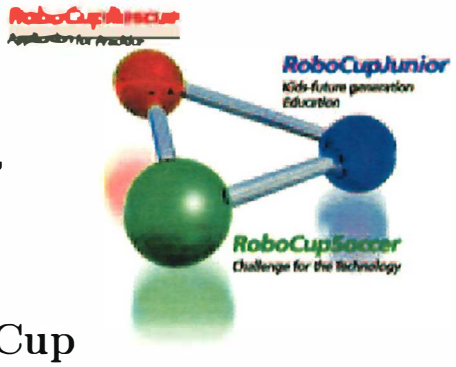
- Le cadre d'exécution d'un agent inclut les propriétés suivantes :
  - La conception : comment l'agent est construit (plate-forme, langage, architecture interne, interactions entre agents)
  - L'infrastructure de communication : la mémoire est-elle partagée ?

Les messages sont-ils point-à-point, multidestinataires ou *broadcasted*? La communication est-elle synchrone ou asynchrone ? etc.
- Le protocole de communication (HTTP, HTML, KQLM, CORBA...)
- La sécurité (présence de marqueurs, authentification...).

- Les propriétés relatives à l'interaction avec l'environnement sont les suivantes :
  - Connaissance : quelle connaissance a l'agent de son environnement ?
  - Prédiction : qu'est-il capable d'en prédire ?
  - Contrôle : quel contrôle peut-il en avoir, que peut-il y modifier ?
  - Historique : les états futurs de l'environnement dépendent-ils des anciens états ?
  - Téléologie : y a-t-il d'autres entités, d'autres agents ?
  - Temps réel : l'environnement risque-t-il de changer pendant que l'agent délibère ?

Chapitre 3

La RoboCup et la RoboCupRescue



3.1 La fédération RoboCup

La Fédération RoboCup est une organisation internationale sans but lucratif enregistrée en Suisse et soutenue par SONY et SGI. Son objectif est de promouvoir la science et la technologie au niveau international, initialement à travers des robots et agents logiciels jouant au football (*soccer*, en anglais) [SGI, 2000, RCS, 2001].

L'histoire de cette initiative remonte au début des années 90. En effet, le tout premier article publié mentionnant l'idée de robots disputant un match de football date de 1992. Son auteur est le professeur Alan Macworth, de l'université British Columbia au Canada. Indépendamment, en 1993, un groupe de chercheurs japonais décida d'organiser le tout premier championnat de football pour robots. Ces chercheurs, issus de deux universités ainsi que du laboratoire de recherche en informatique de SONY, décidèrent de le nommer la «Robot J-League». Ce championnat, alors uniquement national, a rapidement suscité l'intérêt de la part des chercheurs du monde entier, qui demandèrent à y participer. C'est ainsi qu'eut lieu en 1997 à Nagoya le premier championnat international de robots jouant au football, rebaptisé «RoboCup».

Face au succès sans cesse grandissant de la RoboCup, de nouvelles compétitions ont vu le jour. La fédération RoboCup se compose aujourd'hui de 3 pôles : la RoboCup Soccer, la RoboCupRescue[1] et la RoboCup Junior. Ces

[1]Le nom RoboCupRescue s'écrit en un mot et non en deux, suite à des problèmes de marques déposées.

différents pôles se composent eux-mêmes de sous-ensembles spécifiques.

En ce qui concerne la RoboCup Soccer, son objectif aujourd'hui est de créer, d'ici l'année 2050, une équipe de robots capables de battre la meilleure équipe de footballeurs humains. On retrouve dans ce pôle les compétitions suivantes :

- le championnat en mode simulation[2], opposant 2 équipes de 11 agents ;
- le championnat «Small-Size (f-180)», qui fait intervenir 2 équipes de 5 robots dont la surface ne peut dépasser 18×18 centimètres, sur un terrain de 2,4×2,9 mètres ;
- le championnat «Middle-Size (f-2000)», opposant 2 équipes de 11 robots de 2025 centimètres carrés maximum (45×45cm) sur un terrain de 8 à 10 mètres de long et de 4 à 7 mètres de large ;
- les championnats «Humanoid», compétitions avec des robots humanoïdes allant de 40 centimètres de hauteur aux robots humanoïdes «taille réelle», d'un mètre quatre-vingt de hauteur, qui a eu lieu pour la première fois au mois de juin 2002 ;
- le championnat «Sony Four legged Robot», qui fait intervenir les robots-chiens de SONY ;
- le championnat «Commentator Exhibition», qui devrait démarrer dans peu de temps, et dont l'objet sera de créer des commentateurs artificiels de matchs de football.

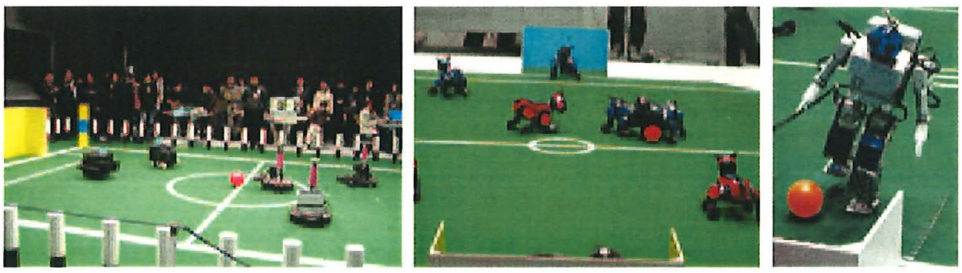


FIG. 3.1 – Différents championnats de gauche à droite : Middle-Size, Sony Four legged Robots et Humanoid.

[2]Le mode simulation fonctionne sans robots. La partie s'observe alors sur un écran d'ordinateur.

Le deuxième pôle des activités de la RoboCup est la RoboCupRescue. Celle-ci oriente ses activités non pas autour du football, mais autour des catastrophes naturelles et autres désastres. Elle se décompose actuellement en deux axes :

- Le premier s'appelle «Rescue Robots». Son principal objectif est l'étude de robots sauveteurs, de leurs capacités individuelles dans des opérations de sauvetage, ainsi que de leurs possibilités de collaboration pour accomplir des tâches spécifiques. Par exemple, on y étudie leur capacité à descendre dans des amoncellements de gravats afin d'y localiser des corps et d'orienter les efforts de sauvetage des équipes de secours qui creusent à la surface. Le premier championnat de cet axe a eu lieu l'année dernière [RCR, 2002]. Notons toutefois que l'utilisation effective de tels robots a déjà eu lieu en situation réelle, notamment dans les décombres des tours du World Trade Center après les tragiques attentats du 11 septembre 2001 [Lee, 2001, Trivedi, 2001, BBC, 2001, CRASAR, 2001].
- Le second axe de la RoboCupRescue est un travail de simulation[3] de catastrophes naturelles. Il se concentre sur les stratégies de planification et de coordination d'équipes de sauvetage. L'objectif des équipes qui prennent part à cette compétition est de minimiser le nombre de morts causés par un tremblement de terre simulé sur ordinateur, à l'aide d'agents représentant des équipes de pompiers, des forces civiles et des équipes paramédicales. L'étude de ce simulateur fut l'objet du stage associé à ce mémoire, et il sera exposé en détail par la suite.

FIG. 3.2 – L'image de gauche est celle d'un robot utilisé dans les décombres des tours de la WTC. Les deux images de droite proviennent du championnat Rescue Robots de 2001.

[3]Tout comme le championnat de simulation de la RoboCup Soccer, celui-ci est sans robot.

Le troisième pôle de la RoboCup, la RoboCup Junior, est orienté quant à lui vers l'éducation. Son objectif est en effet d'éveiller les jeunes à la science et à la technologie à travers 3 compétitions :

- un championnat «Soccer», où deux équipes de deux robots s'affrontent au football ;
- un championnat «Rescue», où un robot doit évoluer sur un terrain à obstacles ;
- un championnat «Dance», dont le but est de faire «danser» un robot sur de la musique, c'est-à-dire d'y associer ses mouvements.

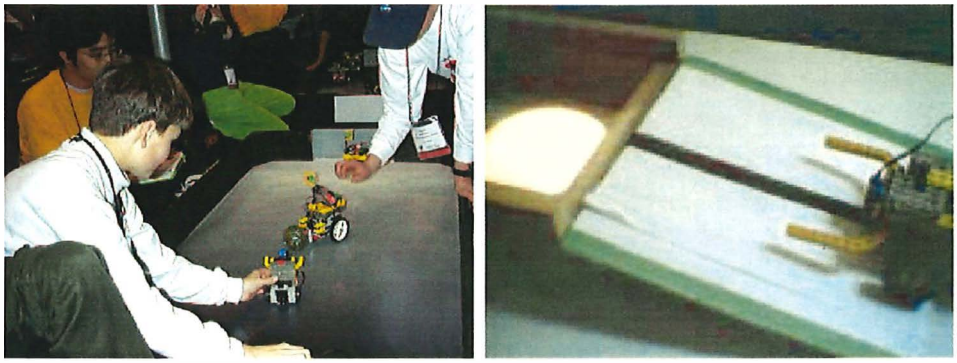


FIG. 3.3 – La photo de gauche provient du championnat Soccer, tandis que celle de droite est issue du championnat Rescue.

3.2 Historique du tremblement de terre

Pour bien comprendre la problématique de la RoboCupRescue, il est nécessaire de faire un retour dans l'histoire du Japon, et plus précisément de revoir en détail la tragique journée du 17 janvier 1995.

Le mardi 17 janvier 1995 à 5h46 du matin, un violent tremblement de terre d'une magnitude de 6.9 sur l'échelle de Richter a frappé la ville de Kobe, faisant plus de 6 000 morts, au moins 300 000 blessés, et rendant 80% des bâtiments complètement inutilisables. Le coût des dégâts dans cette ville d'un million et demi d'habitants dépassa les 300 milliards de dollars US.

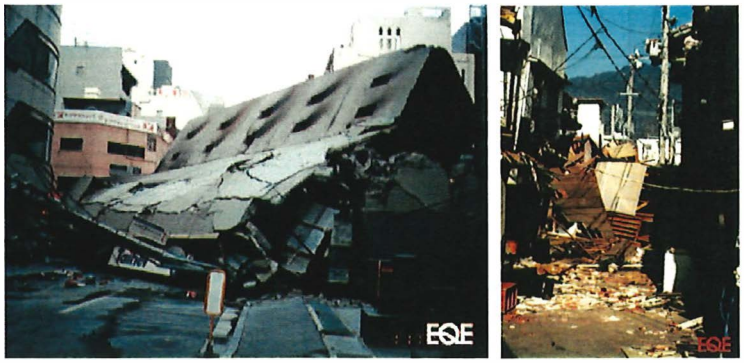


FIG. 3.4 – Bâtiments touchés par le tremblement de terre à Kobe.

FIG. 3.5 – Incendies provoqués par le tremblement de terre à Kobe.

FIG. 3.6 – Les secours, et une autoroute effondrée à Kobe.

Lors de la première secousse, maisons, bâtiments et autres complexes s'effondrèrent, perturbant complètement tous les systèmes de transports publics, et endommageant sérieusement les infrastructures urbaines de base telles que l'électricité, le gaz, l'alimentation en eau et les égouts. Les centres de relais de l'information furent également gravement touchés, empêchant les transmissions concernant les dommages au reste du pays.

De nombreuses victimes furent ensevelies sous les décombres. Plus de 300 brasiers furent rapportés, se propageant largement dans toutes les directions. Les pompiers furent incapables de gérer efficacement la situation, ne trouvant que canalisations brisées et réservoirs fissurés pour s'alimenter en eau. Pire encore, les routes et les aires ouvertes, supposées faire office de coupe-feu, étaient au contraire jonchées de débris en feu projetés des habitations partiellement composées de bois.

Les équipes paramédicales ne purent malheureusement se rendre rapidement sur les lieux où l'on avait le plus besoin d'elles. En effet, elles ont dû faire face à deux problèmes majeurs. Le premier fut bien sûr l'impraticabilité des routes due aux voies crevassées, effondrées, ou jonchées de débris de bâtiments, ou encore à la proximité d'incendies menaçants. Le second, sans doute pire, fut le manque d'information en général. Manque d'information non seulement au niveau du chemin à emprunter pour se déplacer efficacement, mais surtout sur les destinations les plus urgentes ! Les stations de relais de l'information étant partiellement détruites, les équipes ne pouvaient bien souvent compter que sur leurs informations visuelles et auditives pour gérer au mieux la situation.

La surveillance aérienne est arrivée un peu plus tard, et a finalement pu apporter des informations sur la situation globale. Hélas, elle n'a pu apporter que peu de détails, car l'approche de ces unités aériennes provoque beaucoup de bruit et gêne grandement les sauveteurs au sol qui tentent de repérer les voix des victimes ensevelies.

3.3 La RoboCupRescue

Les catastrophes naturelles sont malheureusement des évènements réels qui peuvent nous surprendre à tout moment. De plus, nous ne sommes pas non plus à l'abri d'erreurs humaines pouvant déboucher sur des catastrophes importantes. Il y a donc un réel besoin de développement de stratégies de recherche et de sauvetage optimales dans des désastres à grande échelle. Comme

l'histoire nous l'a appris, bon nombre d'incidents sont au-delà des capacités humaines de prise de décisions sûres. Fournir des systèmes de simulations et d'aide à la décision pourrait améliorer de manière significative la qualité de décision et de compréhension lors de situations d'urgence. Dans le futur, de tels simulateurs pourraient être reliés à des systèmes de communication et d'acquisition de données, mobiles sur le terrain.

D'autre part, il y a un besoin et une opportunité immédiate pour les chercheurs de contribuer à la recherche et au sauvetage en construisant des robots capables de travailler en équipe, chacun d'eux pouvant être spécialisé dans des moyens spécifiques de détection ou de déplacement.

C'est à ces besoins que la RoboCup a voulu répondre en créant son nouveau pôle en 1998 lors des conférences de l'ICRA qui ont eu lieu à Leuven : la RoboCupRescue [Sobek, 1998]. Le développement d'un simulateur de catastrophes naturelles a alors démarré en 1999. L'objectif était d'arriver à créer un simulateur générique pouvant simuler un large domaine de catastrophes, et de l'appliquer à la simulation du tremblement de terre de Kobe.

C'est en l'an 2000 qu'eut lieu la première démonstration publique de ce simulateur, suivie en 2001 du tout premier championnat de la RoboCupRescue. Nous verrons dans les chapitres qui suivent les détails concernant cette simulation et son championnat, mais il nous semble utile de commencer par survoler rapidement son fonctionnement.

La simulation tente de recréer de nombreux aspects de la situation dans laquelle s'est retrouvée la ville de Kobe juste après le tremblement de terre de 1995. Les participants au championnat RoboCupRescue doivent programmer l'«intelligence artificielle» des entités qui vont y représenter les secours[4], que nous appellerons les agents sauveteurs. Ces agents seront alors plongés dans cette situation, ici virtuellement recréée, tel que l'ont été les secours durant le vrai tremblement de terre.

Afin de déterminer objectivement quelle est la meilleure équipe d'agents, plusieurs scénarios de tremblement de terre sont simulés sur ordinateur. Chaque scénario est composé d'une liste de conditions initiales[5], où sont fixées les positions des agents, les endroits de déclaration d'incendie, les rues impra-

[4]Equipes paramédicales, de pompiers et de forces civiles.

[5]Chaque équipe doit préparer deux scénarios avant la compétition, qui serviront pendant le championnat.

ticables, les positions des civils à sauver, etc. Chaque équipe déploie alors ses agents dans chaque scénario[6], où ils doivent gérer au mieux la catastrophe : arriver à déterminer où l'urgence les appelle, négocier le déblayement des routes afin de se rendre efficacement aux endroits stratégiques, déterminer quel tas de gravats doit être prioritairement creusé pour sauver le plus de vies, amener les blessés dans des refuges spécialement aménagés, etc. L'équipe qui aura réussi à sauver le plus d'agents civils et, secondairement, minimisé au mieux les dégâts dûs aux incendies sera déclarée vainqueur.

L'année passée, 7 équipes ont pris part à la compétition, confrontant leurs systèmes d'agents afin de déterminer lequel déployait la meilleure stratégie générale face à la catastrophe simulée. Parmi les pays d'origine de ces équipes, nous retrouvions le Japon, les Etats Unis, l'Iran et l'Australie. Ce fut l'équipe japonaise baptisée «YabAI» qui remporta la victoire.

[6]Pour être exact, chaque équipe est évaluée selon le premier scénario préparé par chacun de ses adversaires, ainsi que sur le sien. Les scénarios secondaires servent pour les demi-finales et la finale.

Chapitre 4

Le simulateur

Dans ce chapitre, nous allons analyser le simulateur utilisé lors du championnat de la RoboCupRescue de 2001[1]. Celui-ci étant l'élément vital et central de la simulation, il nous a paru opportun de le présenter avant même de nous étendre sur ses agents. Nous commencerons par présenter la structure du simulateur, composant par composant. Nous verrons ensuite comment le tout s'articule, s'assemble et fonctionne ensemble.

4.1 Structure du simulateur (Version 0.31)

Le simulateur est un ensemble de modules (programmes indépendants) qui communiquent entre eux grâce à un protocole de communication basé sur UDP[2]. Ces différents modules sont : le noyau (*kernel*), les agents, les simulateurs de composant (incendie, tremblement de terre, embouteillage...), le SIG (Système d'Information Géographique), et les fenêtres de visualisation (*viewers*). En voici la présentation :

- Le **SIG** : le module SIG est celui qui définit et gère toutes les informations nécessaires à la création et à l'actualisation du monde dans la mémoire de l'ordinateur. Le monde qui est reconstruit est une modélisation d'une partie de la ville de Kobe à l'échelle un dixième, et forme l'environnement dans lequel évoluent les agents. Cette modélisation couvre une région de 500 mètres sur 500. On y retrouve ainsi 820 morceaux de routes, 778 groupes de bâtiments, 97 groupes d'individus, etc. Le SIG détient des informations relativement nombreuses et précises sur chaque objet qu'il modélise[3], ainsi que la configuration

[1] Le simulateur est en téléchargement libre à l'adresse [Download, 2001].
[2] UDP : User Datagram Protocol.
[3] La liste complète des informations que le SIG manipule se trouve à l'annexe A.

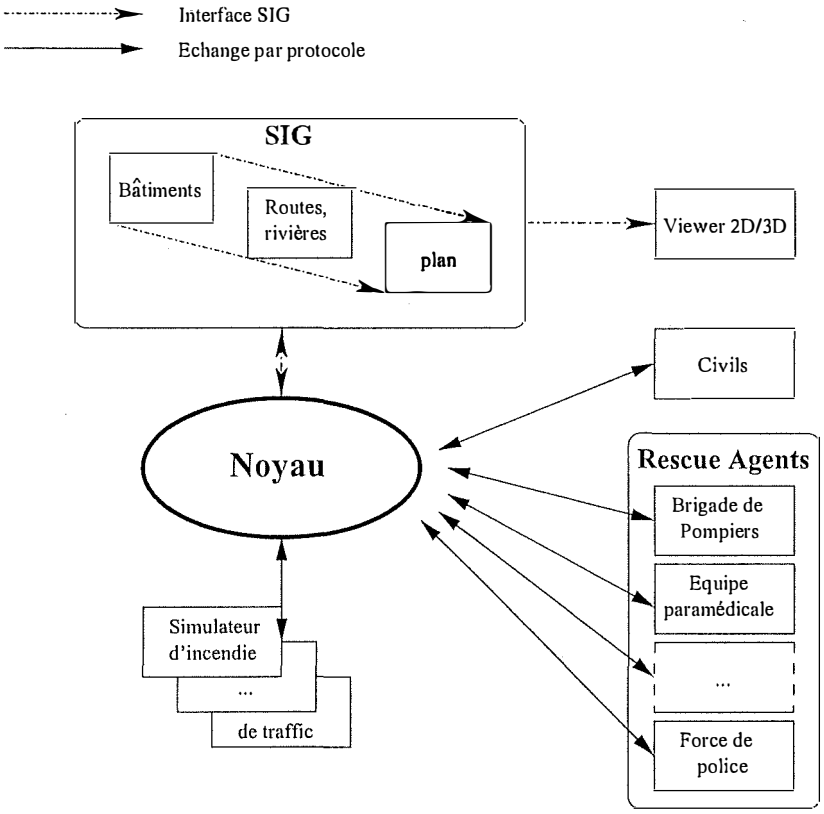


FIG. 4.1 – Schéma des différents modules composant le simulateur.

initiale de ceux-ci. Par exemple, les informations qu'il possède sur les bâtiments sont : la localisation, le nombre d'étages, la matière principale dans laquelle a été fabriqué le bâtiment (bois, trame en acier ou béton renforcé), la superficie, une liste des arêtes qui permettent d'en obtenir une représentation 3D, etc. En ce qui concerne la modélisation des routes, on peut même y trouver des informations telles que la taille des trottoirs ou les périodes des feux lumineux associés aux carrefours. Les rivières ont aussi nombre de détails, la météo y est représentée (force et direction du vent), etc. Les unités de mesure sont souvent de l'ordre du millimètre, les mesures d'angles en secondes, etc. Outre toutes ces manipulations d'informations, le SIG enregistre aussi les détails de la simulation, rendant une analyse postérieure possible.

- Les **agents** : les agents sont aussi des modules du simulateur, puisqu'ils sont nécessaires à la simulation. Ils représentent des individus intelligents qui décident de leurs propres actions d'après les situations dans lesquelles ils sont. Ces agents sont donc des entités virtuelles qui représentent des civils, des pompiers, des commissariats de police[4], etc. Ils forment les programmes clients qui contrôlent les désirs et les actions des individus simulés (le programme serveur étant le «noyau», module présenté un peu plus loin). C'est la seule partie dont doivent s'occuper les participants au championnat, et qui a été développée au cours du stage associé à ce mémoire. Dans le simulateur actuel, un agent représente non pas un individu, mais un groupe de personnes, comme une famille de civils ou une brigade de pompiers. Nous y reviendrons plus loin.

- Les **simulateurs de composant** : ce sont des simulateurs à part entière qui s'occupent de domaines spécifiques de simulation. Ils sont actuellement au nombre de cinq :
  - un simulateur de tremblements de terre ;
  - un simulateur d'incendie ;
  - un simulateur d'accumulation de débris sur les routes ;
  - un simulateur de trafic routier ;
  - un simulateur d'évolution de santé.

  Ces composants calculent ce qui se passera dans le monde en fonction de l'état actuel des éléments dans leur domaine. Notons qu'ils ne détiennent pas d'information du monde à proprement parler, mais manipulent celles du SIG. Ainsi par exemple, le simulateur d'incendie s'occupe de gérer les endroits où il y a des incendies en fonction de la force et de la direction du vent, de la quantité d'eau que les pompiers déversent dessus, de la présence à proximité d'autres éléments (débris, bâtiments, etc.) et de leur type. Cette modularisation a la particularité de permettre assez facilement d'enlever ou d'ajouter différents simulateurs, en fonction de ce que l'on veut simuler. On pourrait par exemple imaginer simuler une tempête de verglas en enlevant simplement le simulateur de tremblement de terre, et en ajoutant un simulateur de verglas (cette hypothèse est développée au chapitre 8).

- Les **viewers** : ce sont des programmes qui permettent de visualiser la simulation. Il y en a de diverses sortes, en 2 ou 3 dimensions, avec

[4]Lorsque l'on dit, par abus de langage, qu'un agent représente un bâtiment, on sous-entend qu'il représente les individus qui l'habitent.

plus ou moins de détails. Vous trouverez des exemples de *viewers* aux figures 4.2, 4.3 et 4.4. Nous décrivons ici de manière approfondie les informations qu'affiche «logViewer», le *viewer* utilisé lors du stage pour la création du système multiagent (figure 4.3). Les bâtiments sont représentés par des rectangles. Leur état varie de sain (en vert) à effondré (en noir), sous l'emprise d'un incendie venant de se déclarer (en jaune) à un incendie vigoureux (en rouge), éteint par les pompiers (de bleu à gris en fonction des dégats causés par l'incendie) ou complètement ravagé par l'incendie (en brun). Les routes sont aussi représentées par des rectangles. Leur état varie de «dégagée» (en blanc) à «complètement obstruée» (en noir). Les agents sont soulignés par une ligne bleue et/ou rouge qui représente leur niveau de santé. On peut repérer un civil (coin supérieur gauche), une équipe de policiers (voiture noire à gauche), une brigade de pompiers (à côté de la goutte d'eau qui représente l'extinction en cours du bâtiment), une équipe paramédicale (à droite)...

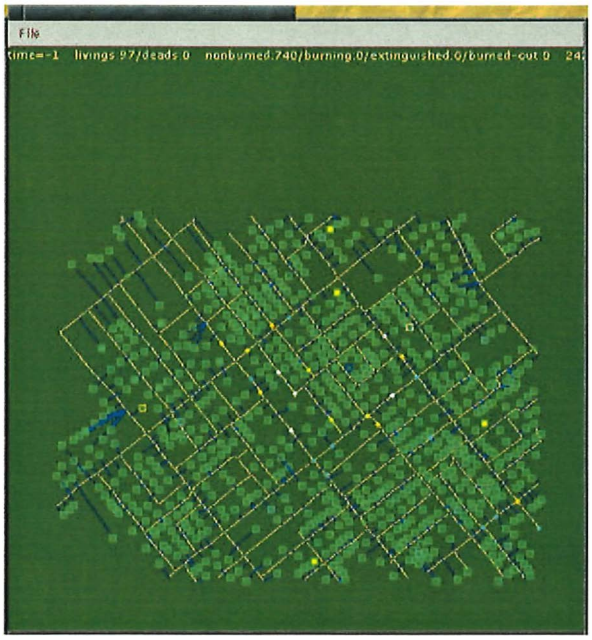


FIG. 4.2 – «`simple2dviewer`», un *viewer* simple et non interactif fourni avec le simulateur [Download, 2001].

FIG. 4.3 – LogViewer, le *viewer* interactif utilisé lors du stage, aussi fourni avec le simulateur [Download, 2001].

FIG. 4.4 – Un *Viewer* en trois dimensions, qui n'a pu être testé lors du stage car son développement n'était pas encore assez avancé (ces images proviennent de [Arc, 2001]).

- Le **noyau** : c'est l'élément central de la simulation. Il en contrôle le processus et organise le partage des informations entre les modules. Il constitue pour l'équipe qui le développe un des défis les plus importants de la construction de ce système de simulation. C'est en effet lui qui devra, dans le futur, gérer des dizaines de milliers de modules[5] et leurs communications en temps réel. Tout échange d'information passe par lui. Ainsi, même lorsqu'un agent désire communiquer avec un autre, il ne peut pas lui envoyer directement son message. Il doit d'abord l'envoyer au noyau, qui, après diverses opérations (telles que des vérifications de validité), le renvoie à l'agent désiré. De même, lorsque le simulateur d'incendie envoie de nouvelles informations sur l'évolution d'un incendie, c'est au noyau qu'il les adresse. C'est seulement alors que celui-ci les renvoie au SIG, ainsi qu'aux autres simulateurs de composants, et aux agents. Il contrôle toute la simulation, tout échange d'informations, et peut vérifier que chaque module remplit correctement son rôle. En outre, ce contrôle lui permet de gérer le synchronisme du système et l'écoulement du temps (ce point est abordé au chapitre 4.2).

- Le **protocole de communication LongUDP** : cet élément n'est pas un module, mais fait bien partie du simulateur dans le sens où il définit les règles de communication et d'échange d'information entre les modules et le noyau. Ce protocole, basé sur le protocole UDP et sur la représentation standard XDR[6], permet une totale indépendance du langage de programmation utilisé pour programmer les agents. C'est pourquoi les agents peuvent aussi bien être programmés en Prolog qu'en Java ou dans n'importe quel autre langage, pour autant qu'ils respectent ce protocole de communication. Ainsi, alors que le noyau et les simulateurs de composants sont développés en C++, les agents développés durant le stage ont été programmés dans le langage Java.

4.2 Le déroulement du temps

Le système simule 5 heures de gestion de la catastrophe après le tremblement de terre. Le simulateur fonctionne en temps discret, et l'unité d'incrémentation du temps utilisée est la minute. La simulation effectue donc 300

[5] Actuellement, le nombre de modules est relativement réduit (107 plus les *viewers*), notamment par le fait que les agents représentent des groupes de personnes. Dans le futur, chaque personne devra être représentée par un agent.

[6] XDR : eXternal Data Representation

cycles, chacun simulant une minute de la catastrophe[7].

Notons, avant de détailler le déroulement de la simulation elle-même, que chacun des simulateurs de composants fonctionne de manière autonome, simulant son domaine à sa manière. En effet, bien qu'ils doivent respecter un délai minimal de réponse pour leur simulation et qu'ils soient tenus d'envoyer régulièrement les résultats pour chaque minute simulée, il n'y a pas de contrainte quant à la façon de faire les calculs. Par exemple, ils peuvent n'utiliser qu'une partie des informations envoyées par le noyau. C'est ainsi que certains simulateurs d'incendie font leurs calculs à partir des données en deux dimensions des bâtiments, alors que d'autres utilisent les trois dimensions. Ils ont en outre la possibilité d'effectuer une simulation continue, ou d'un pas d'incrémentation du temps plus petit. Par exemple, pour des calculs de déplacements de véhicules prenant en compte les accélérations et décélérations, le pas d'une minute est insuffisant.

Chaque agent et chaque simulateur doit avoir rendu ses intentions d'actions ou les résultats de ses calculs dans un délai limité. En ce qui concerne les méthodes de planification des agents, on se retrouve d'une certaine manière avec des contraintes temps réel, puisqu'un mauvais plan d'action vaudra toujours mieux qu'un plan qui arrivera trop tard. En effet, passé un certain délai, le noyau considérera que l'agent n'agit pas, et ignorera ses demandes d'actions pour la minute passée.

4.3 Le déroulement de la simulation

Tout d'abord, le système commence par une initialisation : une fois que tous les simulateurs et les agents sont connectés au noyau, le SIG est le premier à intervenir. Il envoie la configuration initiale du monde au noyau. Celui-ci la fait suivre de manière globale aux simulateurs de composants, et seulement de manière partielle aux agents. En effet, ceux-ci n'ont pas, par exemple, accès aux points initiaux où vont se déclarer les incendies.

Ensuite démarre la simulation. Comme déjà évoqué, le système va simuler 300 minutes, et donc 300 cycles. Voici la description de l'une de ces 300 itérations :

[7]Sur un processeur à 1 gigahertz, une simulation complète dure de l'ordre de 25 minutes réelles.

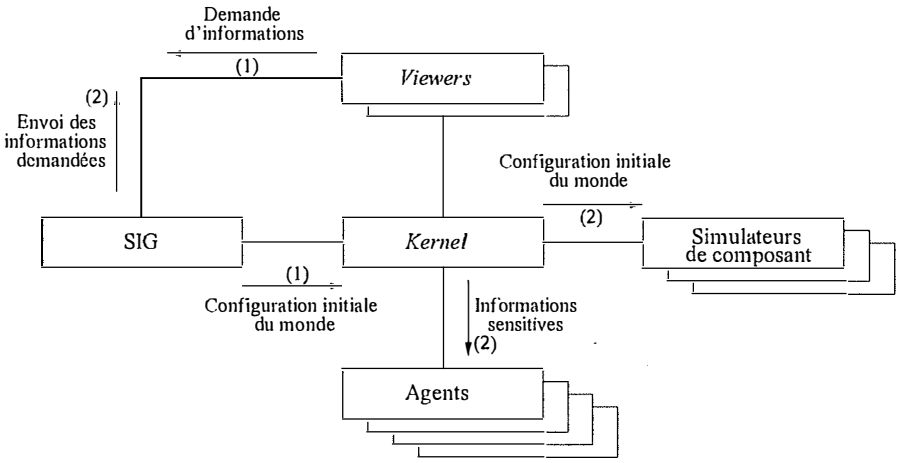


FIG. 4.5 – Communications entre les modules à l'initialisation.

1. Tout d'abord, le noyau envoie à chaque agent ses informations sensitives (visuelles[8] et auditives). L'information visuelle contient des données sur les objets dans un rayon de 10 mètres de l'agent. L'information auditive comprend la parole et les communications radios. Nous détaillerons l'information manipulée par les agents dans le chapitre suivant.
2. Chaque agent, sur base des informations qu'il vient de recevoir, choisit les actions qu'il désire exécuter et envoie ses décisions au noyau.
3. Le noyau récupère les messages venant des agents et les envoie aux simulateurs de composants. Rappelons que ces messages sont filtrés. Par exemple, un message provenant d'un agent représentant un individu mort sera ignoré. Il en sera de même avec les messages qui ne sont pas arrivés dans les délais, qui ne respectent pas le protocole, etc.
4. Sur base des informations reçues, les simulateurs de composants calculent individuellement comment le monde va changer et retournent leurs résultats au noyau.
5. Le noyau intègre les résultats conformes (et arrivés à temps) et les envoie au SIG. Ces résultats sont aussi renvoyés aux simulateurs de composants, afin qu'ils prennent en compte les modifications générales qu'ils ont apportées tous ensemble. Le noyau incrémente alors l'horloge et en informe le ou les *viewer(s)*, afin qu'il(s) puisse(nt) se mettre à jour.

[8]Les agents bâtiments n'ont pas accès à l'information visuelle.

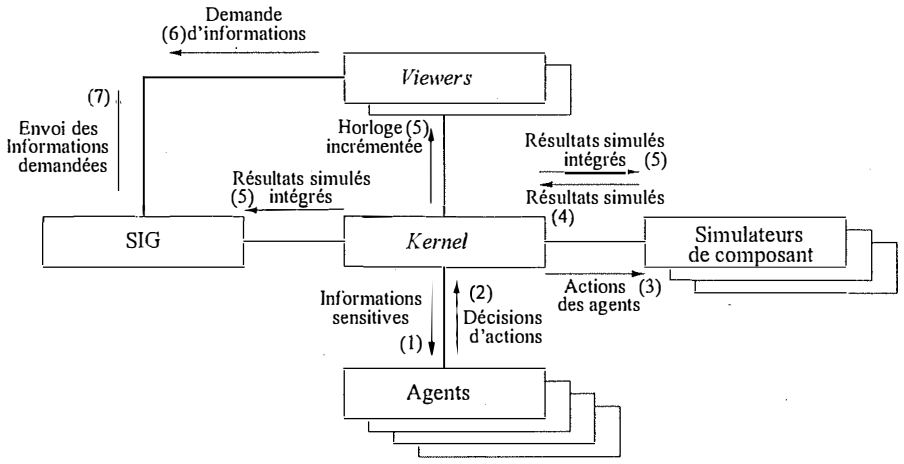


FIG. 4.6 – Communications entre les modules pendant la simulation.

6. Les *viewers* ainsi prévenus du changement de l'horloge demandent au SIG de leur envoyer les modifications qui ont eu lieu dans le monde et affichent ces informations sur l'écran.
7. Le SIG ajoute les modifications du monde à la trace des résultats de la simulation et envoie aux *viewers* l'information demandée.

Ainsi les cycles se succèdent, et ce tant que la simulation n'est pas terminée.

Chapitre 5

Les agents

Nous avons souvent évoqué, au cours des chapitres précédents, les différentes entités de la simulation, les agents, mais sans vraiment entrer dans le détail. Le but de ce chapitre est donc d'examiner ces acteurs en profondeur, d'analyser leurs pouvoirs d'action et les limites de ces pouvoirs, et de dégager les interactions qui peuvent en résulter.

5.1 Les différents agents et leurs actions

Bien que les agents représentent, au sein de la simulation, des regroupements de personnes, leur nombre reste conséquent. Il s'agit de près d'une centaine d'entités qui prennent leurs décisions de manière indépendante et, dans le futur, ce nombre est voué à augmenter encore fortement. En effet, comme nous l'avons déjà évoqué au chapitre 4.1, les versions futures du simulateur devront être capables de gérer plusieurs milliers d'agents, où chaque agent représentera un individu en particulier.

En plus de leur nombre assez élevé, le simulateur doit être capable de gérer leurs différences. Si les individus ne sont pas encore distingués entre eux, les groupes d'individus ont certaines spécificités qui les distinguent les uns des autres. Voici une description des différents types d'agents qu'on retrouve dans la version actuelle du simulateur :

- **Les familles de civils** : ces agents, au nombre de 72, sont les seuls à être fournis avec le simulateur. Le comité organisateur se charge de les réaliser en leur donnant des comportements type dont les participants devront tenir compte. Leur comportement est actuellement assez simple, mais devrait évoluer rapidement au fil des versions futures. Dix

de ces 72 agents sont en voiture et se déplacent à une vitesse plus élevée (60 km/h environ, contre 4 km/h pour les piétons). Leur comportement se limite à crier à l'aide lorsqu'ils sont blessés, et à déambuler dans les rues de manière désordonnée. Dans le futur, ils auront des réactions plus cognitives, telles que retourner chercher des objets personnels dans les bâtiments, prévenir les secours de la présence de blessés, les assister dans leur travail de sauvetage, etc.

- **Les brigades de pompiers** : ces brigades, au nombre de dix, sont chargées d'éteindre les incendies. Pour y parvenir, elles doivent se déplacer avec leur camion et trouver une bouche d'eau avec un débit suffisant pour atteindre leur cible. Dans les versions 0.x du simulateur, les pompiers tirent l'eau en débit constant et en quantité illimitée directement de leur camion, mais il ne s'agit que d'une simplification temporaire qui devrait disparaître d'ici un an. Ils sont en contact radio entre eux et avec leur caserne.

- **Les forces de police** : de l'anglais, «*police force*», il s'agit plus d' équipes de protection civile que de policiers. Ces équipes sont aussi au nombre de dix et sont chargées de déblayer les routes des débris qui les obstruent à l'aide de bulldozers et autres machines, et de créer des voies praticables pour les autres agents. Ils sont en contact radio entre eux et avec leur commissariat de police.

- **Les équipes d'ambulances** : de l'anglais, «*ambulance team*», il s'agit en fait de cinq équipes paramédicales qui se chargent non seulement de soigner les blessés et de les amener en ambulance dans des hôpitaux ou dans des refuges, mais aussi d'aller creuser dans les décombres des bâtiments pour en extraire les personnes qui y sont enfouies. Dans la version actuelle du simulateur, ces équipes ont accès à une information quelque peu irréaliste : elles connaissent le temps exact nécessaire pour un sauvetage ainsi que l'état de santé des civils qui sont enfouis sous les gravats... Les équipes sont en contact radio entre elles et avec leur centrale.

- **Les bâtiments** : les agents qui représentent des bâtiments sont un peu particuliers. En effet, ils représentent non pas la structure elle-même, mais le personnel qui l'habite. On en compte actuellement trois, un par type d'agent sauveteur : **une caserne de pompiers**, **un commissariat de police**, et **un centre paramédical**. Un agent bâtiment n'a accès qu'à l'information auditive qui lui parvient soit des radios des

agents qui en dépendent, soit des autres agents bâtiments. Ils servent de centralisateurs d'information et de coordinateurs entre tous les agents sauveteurs. Dans la version actuelle du simulateur, la destruction d'un bâtiment représenté par un agent n'a pas d'incidence sur les capacités d'actions de cet agent.

On peut résumer les actions que peuvent exécuter les agents en deux classes : celles communes à tous les agents et celles spécialisées, disponibles seulement pour certains agents.

- Les actions communes sont les suivantes :
  - se déplacer (sauf pour les bâtiments) ;
  - parler à voix haute à un agent à proximité ;
  - communiquer à l'aide d'une radio avec les agents et le centre de même type ;
  - ne rien faire.

- En ce qui concerne les actions spécialisées :
  - les agents pompiers peuvent éteindre un incendie ;
  - les agents policiers peuvent déblayer les routes ;
  - les agents paramédicaux peuvent déterrer ou transporter d'autres agents (civils ou sauveteurs) ;
  - les agents centres peuvent communiquer avec les autres agents centres.

5.2 Limitations imposées

Une simulation repose sur une modélisation de la réalité, c'est-à-dire sur une représentation abstraite de certains éléments de cette réalité. Afin de tenir compte des limitations physiques des intervenants réels, les organisateurs de la RoboCupRescue ont imposé diverses limitations au comportement des agents.

Pour rappel, l'unité d'incrémentation du temps est la minute. C'est donc elle qui va régir le nombre d'actions que pourra exécuter un agent en fonction du temps.

Ainsi, un agent a la possibilité, en une minute :

- d'émettre quatre messages ;
- d'écouter quatre messages ;
- de réaliser une autre action

Les messages émis ou reçus peuvent être vocaux ou radios. Ils ne sont pas ciblés vers un destinataire mais sont émis pour tout ceux qui peuvent l'entendre. Les messages radios sont envoyés à tous les agents du même type que celui de l'émetteur, et les messages vocaux sont, eux, envoyés à tous ceux étant à moins de 30 mètres de l'émetteur.

Comme les agents n'ont le droit d'écouter que quatre messages sur l'ensemble de ceux qui leur parviennent, ils doivent pouvoir sélectionner ceux qu'ils désirent entendre. Cette sélection doit se faire en fonction de l'identité de l'émetteur (sur base de son identifiant unique). Il n'est donc pas possible pour un agent d'écouter le début d'un message pour décider s'il est concerné, puisque sa seule source de sélection est «qui a émis le message», et non «à qui il est adressé».

Il existe également une liste de limitations physiques «naturelles» auxquelles sont soumis les agents. Parmi celles-ci, on retrouve :

- le champ de vision[1] : 10 mètres ;
- la portée de la voix : 30 mètres ;
- la portée des lances à eau : 30 mètres ;
- le débit maximal des lances à eau[2] : $1 m^3/min$;
- la vitesse des piétons : 1.2 m/s (4.3 km/h)
- la vitesse des véhicules : 16 m/s (57.6 km/h)
- etc.

5.3 La collaboration entre agents

Les agents, pris isolément, ont une capacité d'action limitée. En effet, un agent policier peut déblayer des routes mais ne peut pas venir en aide aux blessés. De même, une brigade de pompiers ne peut éteindre un incendie que si elle peut s'en approcher suffisamment. Son efficacité dépend donc de l'état du réseau routier et des agents policiers. Il est aussi préférable pour une équipe paramédicale que les décombres dans lesquels elle creuse ne soient pas situés en plein incendie...

[1] Les incendies sont les seules manifestations visuelles qui soient accessibles à tous, quel que soit l'endroit où se trouve l'agent ou l'importance de l'incendie.

[2] Rappelons que dans les versions 0.x du simulateur, les pompiers tirent l'eau en quantité illimitée et débit constant directement à partir de leur camion.

On peut imaginer bien d'autres situations, pour montrer à quel point les agents ont besoin de collaborer s'ils veulent arriver à réaliser leur objectif commun : minimiser en priorité les pertes humaines et, si possible, les pertes matérielles. D'une manière générale, on peut dégager deux types de collaboration entre agents de la RoboCupRescue. Le premier type est une collaboration entre agents de même sorte. Bien que ce ne soit pas toujours le cas, on peut généralement considérer qu'un groupe d'agents de même type travaille plus rapidement qu'un agent pris isolément. Par exemple, lorsqu'il s'agit d'éteindre un incendie violent, plusieurs camions de pompiers sont plus efficaces qu'un seul. Il est cependant aussi vrai que pour éteindre de tout petits incendies, plusieurs groupes de pompiers risquent d'être inutiles, voire de se gêner. Le second type est une collaboration entre agents de différentes sortes. Comme nous le verrons plus loin, ce type de collaboration est plus difficile à mettre en oeuvre, à cause du nombre d'échanges de messages qu'elle nécessite et du goulot d'étranglement causé par les agents bâtiments (voir point 5.4). Ce type de collaboration est pourtant nécessaire et même vital puisque les aptitudes des agents sont complémentaires. Nous présentons ici une liste non exhaustive d'exemples où la mise en oeuvre d'un tel type de collaboration améliore sensiblement la situation :

- les agents policiers déblayent la route pour optimiser les efforts des autres agents afin que ceux-ci puissent mener leurs buts à bien ;
- les agents pompiers s'occupent d'éteindre les incendies risquant de miner le travail des équipes paramédicales ;
- tous les agents peuvent s'échanger diverses informations pour optimiser leur travail respectif :
  - l'emplacement de civils blessés ou enterrés ;
  - l'emplacement de routes obstruées ;
  - etc.
- les agents centres récoltent et centralisent les informations de manière à dégager une vision globale du désastre et en assurer efficacement la gestion.

5.4 La communication

Comme nous l'avons déjà souligné, la communication occupe une place primordiale dans l'organisation des secours. Nous avons vu qu'il existait deux types de communication : la communication vocale et la communication radio. La première est très peu utilisée car elle demande aux agents d'être relativement proches pour dialoguer, à savoir moins de 30 mètres, ce qui est

difficilement réalisable en l'état actuel du simulateur[3].

La communication radio est beaucoup plus utilisée mais demande néanmoins une mise en oeuvre plus subtile lorsqu'on veut l'utiliser pour dialoguer entre agents de types différents. D'une manière générale, les messages échangés suivent le schéma présenté à la figure 5.1.

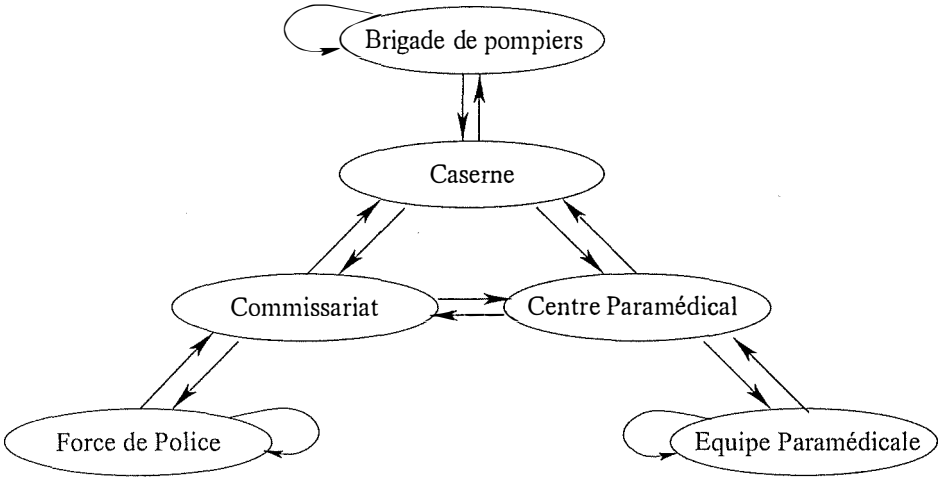


FIG. 5.1 – Schéma général des communications radio.

Ce schéma nous permet de déduire le temps minimal nécessaire à un message pour atteindre sa destination. Prenons comme exemple le cas d'une brigade de pompiers qui désire renseigner une équipe paramédicale sur la présence d'un blessé. Dans un tel scénario, 3 minutes sont nécessaires au minimum pour que le message parvienne à son destinataire :

- **T=0**. La brigade émet son message par radio.
- **T=1**. La caserne (et les brigades de pompiers) reçoit le message, l'analyse et le retransmet grâce à son système radio.
- **T=2**. Le centre paramédical (ainsi que le commissariat et les brigades de pompiers) reçoit le message et le retransmet grâce à son système radio.
- **T=3**. Les équipes paramédicales (ainsi que le commissariat et la caserne) reçoivent le message, qui atteint enfin sa destination.

[3]La simulation se déroulant par cycles d'une minute, il est presque impossible pour deux agents de se suivre, et donc d'établir un dialogue. Ceci est présenté plus en détail au chapitre 7 concernant les critiques sur la simulation.

Si l'on examine ce schéma en parallèle aux limitations imposées (cf. page 39), on constate que les centres forment un goulot d'étranglement au niveau des échanges de messages. Illustrons ce propos avec le scénario suivant (figure 5.2) :

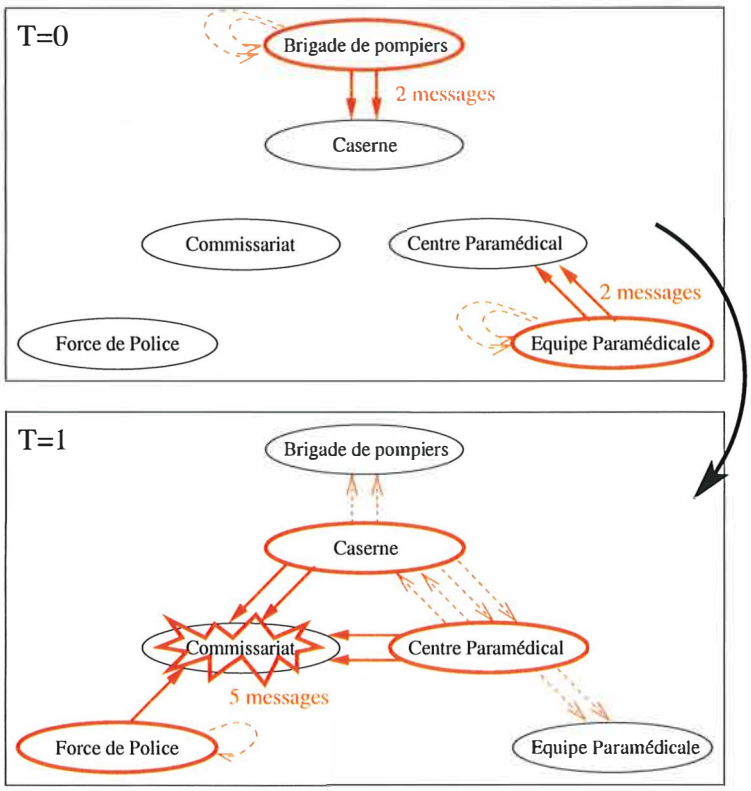


FIG. 5.2 – Exemple de surcharge de messages radios.

- **T=0**. Deux brigades de pompiers émettent une demande de déblayage de routes (à destination des forces de police) par radio. Deux équipes paramédicales font de même.
- **T=1**. La caserne (et les brigades de pompiers) reçoit les deux messages provenant des brigades. Le centre paramédical (ainsi que les équipes paramédicales) reçoit les deux messages des équipes paramédicales. La caserne et le centre paramédical transmettent chacun leur message respectif sur leur système radio. A ce même moment, un agent policier émet un message pour signaler un blessé.

> Concentrons-nous maintenant sur les messages que reçoit le commissariat de police : deux messages de la caserne, deux du centre paramédical, et un d'une équipe de police. Cela nous fait au total 5 messages. Or, comme spécifié dans les limitations, un agent ne peut écouter que 4 messages maximum par minute, ce qui implique qu'un message sera forcément perdu !

On voit aisément que le système de communication imposé par le simulateur ne donne guère la possibilité de créer des stratégies dynamiques élaborées par dialogue entre les différentes entités, puisqu'il suffit de 5 communications pour engorger tout le système[4]. Or, nous avons 28 agents susceptibles d'envoyer à tout moment jusqu'à 4 messages différents par minute, auxquels il faut encore ajouter les messages vocaux d'appels à l'aide des agents civils !

On voit nettement ici la nécessité de mettre en place un système élaboré de gestion d'émissions et de traitement des messages pour éviter soit d'en perdre (trop), soit d'en voir fortement retardés dans des tampons sans cesse grandissants, ce qui pourrait bien impliquer pour certains messages de ne jamais arriver à leur destination dans des délais raisonnables...

5.5 Classification des agents

Nous proposons maintenant de classer ces agents[5] ainsi que le système multiagent lui même en fonction de leurs propriétés et des classes proposées aux chapitres 2.1.2 et 2.2.1 (pages 13 et 17).

[4] Cette limitation fut l'objet de nombreuses discussions afin de déterminer si elle était réaliste pour les centres. Nous en reparlerons au chapitre 7.

[5] Rappelons que notre étude s'est portée plus particulièrement sur les agents sauveteurs, les agents civils étant quant à eux fournis par le comité organisateur de la RoboCupRescue. La classification que nous proposons se concentre donc sur ces premiers, bien qu'elle puisse généralement s'appliquer aux agents civils.

5.5.1 Propriétés intrinsèques des agents

Propriétés	Valeurs
Durée de vie	longue (permanente)
Architecture	hybride
Construction	libre
Mobilité	fixe
Adaptabilité	ajustée (autodidacte ?)

Nous avons retenu pour la durée de vie des agents la valeur «longue», puisqu'ils sont présents du début à la fin de la simulation. Même lorsqu'ils représentent des entités décédées, ils sont encore présents dans le système, bien que leur possibilité d'action soit nulle.

Le type de l'architecture mérite une explication un peu plus approfondie, puisque nous n'avons pas encore parlé de la réalisation de nos agents. Bien qu'il soit tout à fait possible de rencontrer des agents purement réactifs ou purement cognitifs dans la simulation, les architectures sont généralement plutôt hybrides, combinant le réactif et le délibératif en fonction des situations qui peuvent aller des plus urgentes au moins urgentes. Nous pensons par ailleurs que caractériser l'architecture en tant que purement réactive ne correspondrait pas à l'esprit de la simulation. En effet, l'envoi des informations sensitives du noyau aux agents leur permet d'élaborer une base de connaissance sur leur environnement, ce qui représente déjà une caractéristique d'agents cognitifs. Ensuite, les agents peuvent communiquer entre eux selon des protocoles plus ou moins évolués et coopérer volontairement (par opposition à une coopération émergente) afin d'améliorer leurs performances. Nous pensons qu'il s'agit ici aussi de caractéristiques plutôt cognitives. En ce qui concerne les agents que nous avons réalisés, nous le verrons en détail dans le chapitre suivant, ils se servent de règles d'action assez simples, sans élaboration de plans. Ce procédé de décision correspond plutôt à la classe des agents réactifs. Cependant, ces règles tiennent compte de leurs connaissances, de buts de plus haut niveau, de priorités sur ces buts en fonction de l'état de l'environnement - même lointain - des autres agents, des demandes de coopérations communiquées par ces autres agents... Comme vous pouvez le voir, il serait sans doute assez maladroit de classer les agents exclusivement dans une classe ou dans l'autre. C'est pourquoi nous pensons qu'ils ont des propriétés de ces deux classes à la fois et que nous avons caractérisé l'architecture d'hybride.

Nous avons vu que la modularité du simulateur permettait une totale indépendance du langage utilisé. Ainsi la construction des agents est totalement libre. En ce qui concerne celle de nos agents, nous avons utilisé uniquement le langage Java, ce qui lui confère un caractère procédural. Bien que nous ayons utilisé un système de règles de production (lui-même écrit en Java), celui-ci n'est pas omniprésent dans le code.

Nous avons déterminé la mobilité comme fixe, puisque les agents ne se déplacent pas d'un système à un autre. Ils restent dans le simulateur pour lequel ils sont construits et ne sont pas censés en sortir (le simulateur est d'ailleurs tel qu'ils ne sauraient plus y rentrer une fois en dehors).

Avant de caractériser l'adaptabilité, il est nécessaire de rappeler que les agents évoluent dans un environnement rapidement changeant (expansions des incendies, aggravation de l'état de santé de certains civils, etc.). Il est donc fortement improbable que les agents puissent être efficaces s'ils ne sont pas capables de s'adapter aux changements de leur environnement[6]. C'est pourquoi nous avons retenu la valeur de l'adaptabilité «ajustée». Pourrait-elle être autodidacte ? A priori, nous ne voyons rien qui puisse l'empêcher, sinon qu'il faut alors entraîner ces agents sur plusieurs simulations. Une seule simulation semble trop courte pour qu'un agent déduise les réflexes nécessaires aux situations qui se présenteront alors à lui. L'adaptabilité des agents va donc dépendre des algorithmes que leur concepteur va mettre en oeuvre. Il est nécessaire que les agents puissent s'adapter, mais qu'ils le fassent par ajustements prévus et/ou par apprentissage dépend uniquement des décisions de leur programmateur.

5.5.2 Propriétés extrinsèques des agents

Propriétés	Valeurs
Endroit	distant
Autonomie sociale	indépendante (contrôlée ?)
Sociabilité	conscient, responsable
Bienveillance	coopératif (entraide ?)
Interaction	Logistique : direct et indirect Sémantique : libre (déclarative ?)

La modularité du simulateur prévoit des exécutions délocalisées de tous les modules. On pourrait bien entendu lancer tous les modules d'un même

[6] Cette remarque concerne sans doute moins les agents civils.

endroit et obtenir des agents qui seraient lancés localement, mais initialement ces agents sont réalisés pour participer au championnat, auquel cas ils doivent pouvoir être exécutés à distance.

Le cas de l'autonomie sociale est plus difficile à trancher, puisque les agents sauveteurs peuvent être contrôlés par les centres. Tout dépend du système que l'on a réalisé, même si les limitations et les problèmes de communication présentés au chapitre 5.4 rendent le contrôle moins probable. Dans notre cas, les agents centres n'ont qu'un rôle de relais d'information, rendant l'autonomie sociale de nos agents indépendante.

Concernant la sociabilité, nous avons présenté au chapitre 5.3 la nécessité d'un minimum de coopération des agents lors de la simulation. De ce fait, les agents ne devraient pas être autistes. Aussi les qualifierons-nous d'agents conscients. Au niveau de leur responsabilité, comme les agents ont chacun leur tâche spécifique dont ils doivent s'occuper, ils en sont aussi responsables car ils sont bien «engagés» dans ce qu'ils font. En effet, nous pensons pouvoir dire, par exemple, que les agents pompiers sont responsables d'empêcher les incendies de s'étendre et de les éteindre. Le même raisonnement peut être appliqué pour chacun des autres types d'agents[7].

La bienveillance des agents devrait être au minimum coopérative. L'objectif des agents sauveteurs étant de minimiser les pertes humaines, si les agents devaient être antagonistes, ils risqueraient de se gêner. Une compétition générale entre les agents n'est sans doute pas non plus souhaitable, encore qu'elle puisse éventuellement être exploitée de manière à déterminer les meilleures coalitions d'agents, les meilleurs plans d'actions, etc. Nous avons proposé une bienveillance coopérative «au minimum», parce que nous pensons qu'elle devrait même aller jusqu'à l'entraide. Un agent devrait par exemple pouvoir délaisser sa tâche pour aller en aider un autre. C'est d'ailleurs le cas pour certains des agents que nous avons réalisés.

En ce qui concerne les interactions, du côté logistique, nous retrouvons les deux types (direct et via des intermédiaires), puisqu'il existe les communications vocales qui sont directes et les communications radios entre agents de types différents qui, elles, passent par l'intermédiaire des centres[8]. Du côté sémantique, le format d'échange entre les agents était libre lors du championnat de 2001. Cependant, tout le reste du système communiquait à l'aide

[7]Exception faite des agents civils.

[8]Caserne de pompiers, commissariat de police et centre paramédical.

du format de représentation déclaratif XDR[9]. De plus, les propositions de futures régularisations des échanges de messages entre agents semblaient se diriger vers le même type de resprésentation déclarative. En ce qui concerne nos agents, ils utilisent un langage du même type, et donc leurs interactions sémantiques sont déclaratives.

5.5.3 Propriétés du système multiagent

Il nous reste à donner les propriétés du système multiagent lui-même :

Propriétés	Valeurs
Unicité	homogène
Granulosité	fine
Structure de contrôle	hiérarchique
Autonomie d'interaction	XDR, LongUDP, limite d'échange
Autonomie d'exécution	contrôlée

Nous pensons que ce système multiagent est relativement homogène, bien que les agents ne soient pas tous identiques. Si chaque type d'agent possède une ou deux actions différentes des autres types, ils sont pourtant identiques au sein d'un même groupe et se ressemblent énormément entre groupes différents : même perception de leur environnement, mêmes capacités de déplacement, mêmes moyens et ontologie de communication, etc. Les centres sont sans doute les seuls agents à s'en écarter un peu mais, ici encore, ils ressemblent aux précédents avec simplement moins de capacités qu'eux[10].

La valeur que nous avons affectée à la propriété granulosité peut sans doute être controversée. Il est vrai que le niveau de détail est toujours fin ou gros *par rapport à* quelque chose. Nous devons donc nous rapporter à ce qui nous concerne ici : la gestion d'une catastrophe naturelle à l'échelle d'une ville. Nous pensons que pour les besoins de la simulation, et en l'état où elle est actuellement, la taille des trottoirs, le nombre de bandes de circulation, la liste des arêtes des bâtiments, la présence de feux de signalisation, etc. sont autant de détails qui, même s'ils apportent des précisions intéressantes dans le domaine, ne sont pas obligatoirement nécessaires à la simulation. C'est la

[9]Rapellons qu'il s'agit d'une suite d'entiers, dont le premier est un code significatif représentant une propriété, un type d'agent, une action, ... Les entiers suivants sont les identifiants des objets concernés par le code significatif.

[10]Le cas des agents civils rentre dans le même raisonnement : ils n'ont simplement pas de pouvoir d'action sur leur environnement.

raison qui nous fait penser que la granulosité du système est relativement fine. Bien sûr, on pourrait encore pousser le niveau de détail plus loin, mais ces détails ne seraient sans doute plus justifiables.

La structure de contrôle est dans ce cas-ci clairement hiérarchique. Lorsque nous avons étudié le fonctionnement du simulateur, nous avons mis en évidence son fonctionnement discret, qui oblige les agents à agir selon une certaine temporisation. De plus, nous avons présenté dans ce chapitre les limitations qu'impose le système à ses agents. Le noyau se pose donc en maître hiérarchique du système multiagent, puisqu'il contrôle les agents, leurs communications et leurs actions.

Nous ne désirons pas entrer dans les détails de la propriété «autonomie d'interaction», car elle concerne l'utilisation de l'interface dont doivent se servir les agents pour se connecter, pour dialoguer. Rappelons juste la présence du protocole de communication LongUDP et du format de représentation XDR que tous les agents doivent respecter, ainsi que la limite du nombre de messages que les agents peuvent s'échanger par cycle.

Enfin, l'autonomie d'exécution est contrôlée par le fonctionnement par cycle du simulateur. En effet, comme nous l'avons vu, un agent peut faire une action et envoyer et recevoir quatre messages en un cycle. Cependant, il n'y a pas de sous-cycle à ce cycle. Le simulateur ne fera pas de différence entre un agent qui décide d'envoyer un message toutes les 15 secondes simulées et un autre agent qui envoie ses quatre messages en même temps. De même, un agent n'a aucun intérêt à déterrer un civil en début de cycle plutôt qu'en fin. Le simulateur récupère toutes les demandes d'actions, et seulement à un moment bien précis de son cycle, il contrôle leur validité et les intègre toutes comme si elles s'étaient passées exactement en même temps. Rappelons aussi qu'un acte émis trop tard ne sera pas considéré. L'exécution d'un agent est donc bien contrôlée par le simulateur.

Chapitre 6

Mise en oeuvre d'un SMA

L'objectif de ce chapitre est de présenter le système multiagent (SMA) que nous avons réalisé lors du stage attaché à ce mémoire. Notre désir étant de nous orienter plutôt vers des problèmes cognitifs que techniques, nous avons décidé de le baser sur le système multiagent vainqueur du premier championnat en 2001. Nous allons commencer ce chapitre en revenant brièvement sur ce championnat. Ensuite, nous présenterons le système multiagent de l'équipe gagnante, ainsi que le nôtre. Enfin, nous terminerons par l'étude de nos résultats par rapport aux résultats de l'équipe championne.

6.1 Le championnat

Avant de commencer, nous désirons rappeler le fonctionnement du championnat. Il s'agit d'une simulation d'un tremblement de terre, où sont recréés certains aspects importants de la catastrophe, à savoir : incendies et effondrements de bâtiments, entravant de nombreuses voies de circulation, ensevelissant de nombreux civils, provoquant des embouteillages, etc. Le but des agents sauveteurs est de sauver prioritairement les vies humaines (agents civils blessés, enterrés ou gravement brûlés) et secondairement de limiter les dégâts matériels causés par les incendies.

Les équipes sont évaluées selon la fonction suivante, qu'il s'agit de minimiser :

$$V = nombre\ de\ morts + \frac{I * D}{(B_{init} * H_{init})}$$

Avec :

- B_{init} la surface totale des bâtiments : $B_{init} = 148\,951\,m^2$
- H_{init} le nombre de points de vie total[1] : $H_{init} = 970\,000$
- I la surface incendiée : $I = B_{init} - B_{final}$
- D le nombre de blessures (points de vie perdus) : $D = H_{init} - H_{final}$

Notons que le deuxième terme est compris entre 0 et 1 et donc ne sert qu'à départager deux équipes ayant sauvé le même nombre de civils. Les résultats du championnat de 2001 sont reproduits à titre d'exemple à l'annexe B.

Les équipes sont testées plusieurs fois sur la même carte, qui représente Kobe, mais suivant diverses situations initiales préparées par ses adversaires. Entre deux situations différentes, certains facteurs peuvent varier : la position initiale des agents (sauveteurs et civils) ainsi que la position et le nombre des foyers d'incendies.

6.2 Le système multiagent YabAI

Les systèmes multiagents étant avant tout des systèmes distribués, la première chose à réaliser lorsqu'on veut développer un groupe d'agents est l'étude du protocole de communication et la mise en oeuvre d'interfaces qui vont permettre aux agents de se connecter et de se faire comprendre par le(s) système(s) dans le(s)quel(s) ils vont rentrer. L'objet de notre étude n'étant pas les interfaces de communications d'un système distribué, nous avons préféré faire abstraction de tous les problèmes qui sont liés à ce domaine pour nous orienter plus particulièrement vers ceux qui sont propres aux agents.

C'est pourquoi nous avons décidé de ne pas réaliser complètement nos agents depuis le début et de récupérer des agents ayant participés à la compétition précédente. Nous avons choisis l'équipe YabAI non seulement pour sa remarquable performance lors du championnat de 2001, mais aussi pour son code source écrit en Java et libre d'utilisation [Takeshi, 2001] . Le code

[1] Nous avons 72 agents civils + 10 brigades de pompiers + 10 équipes de policiers + 5 équipes paramédicales = 97 agents. Chaque agent démarre avec 10 000 points de vie, ce qui nous donne un total de 970 000 points de vie. Les agents centres n'ont pas de points de vie, ils interviennent dans la surface totale des bâtiments.

comporte des commentaires en japonais, aussi avons nous réalisé une documentation explicative du code source de ces agents, dans une structure se rapprochant de la «JavaDoc», et que vous trouverez à l'annexe C.

Voici les différentes classes offertes par cette équipe :

1.	Predicate	Function	Domain Kd'Tree
2.	Constants	RescueObject	ObjectPool
3.	Route	Router	CostFunction
4.	IO		
5.	Message	Action	History
6.	Main	Controller	Scope
7.	FireBrigadeController PoliceForceController		AmbulanceTeamController CivilianController
8.	FireStationController PoliceOfficeController		AmbulanceCenterController

FIG. 6.1 – Liste des classes offertes par le code YabAI

1. Les 4 premières classes mettent en oeuvre toutes sortes d'outils tels que la théorie des prédicats du premier ordre, l'utilisation de fonctions, d'ensembles particuliers et d'arbres d'objets.
2. La première des trois classes suivantes est composée de toutes les valeurs constantes, allant des codes utilisés pour le protocole de communication à la définition des limites imposées par le comité. La seconde classe contient les propriétés et la hiérarchie[2] des objets qui vont évoluer tout au long de la simulation. C'est notamment dans cette classe que sont retranscrites toutes les informations du SIG (annexe A). La dernière classe définit la base de connaissance dont va disposer chaque agent.
3. Les trois classes de ce point permettent de déterminer un chemin de poids minimum dans le graphe représentant le réseau routier. La capacité de se déplacer efficacement est très importante pour les agents, spécialement lorsqu'ils doivent secourir des civils gravement blessés, ou encore prévenir l'expansion d'un incendie qui vient de se déclarer ! La classe CostFunction définit plusieurs fonctions de détermination des poids. Elle permet par exemple de calculer simplement les distances à vol d'oiseau. Mais elle permet surtout de prendre en compte l'état des

[2]Nous vous invitons à consulter le schéma page 133 afin de vous faire une idée de la structure qui modélise le monde.

routes, selon la fonction suivante :

$$coût = longueur * \begin{cases} 1 & \textit{dégagée} \\ 10 & \textit{non confirmée} \\ 100 & \textit{encombrée} \end{cases}$$

Chaque morceau de route, ou encore chaque arrête du graphe, est étiqueté par le poids donné par cette fonction et permet ainsi d'utiliser les itinéraires les plus sûrs.

4. La classe IO gère les échanges de données et les communications entre les agents et le noyau. Elle permet de faire abstraction du protocole de communication.
5. La classe Message définit les messages que vont s'échanger les agents. Ils sont composés de deux entiers, le premier étant un code signifiant par exemple «blessé léger ici», et le deuxième l'identifiant de l'objet concerné. La classe History sert de base de données sur les actions exécutées par l'agent, et la classe Action définit la structure de ces données (un nom d'action et l'heure de réalisation).
6. La classe Main sert à lancer l'exécution des agents. La classe Controller, elle, permet le contrôle générique des agents. Elle est située entre la couche IO et celle où l'agent prend ses décisions. Il n'y a donc aucune prise de décision dans cette classe mais tout un jeu de méthodes servant, par exemple, à maintenir à jour la base de connaissance de l'agent au fil de son exécution. Notons deux techniques utilisées à ce niveau pour faire des inférences sur les informations reçues par le noyau : premièrement, les agents observent d'où viennent les autres agents présents dans leur champ de vision et font des déductions quant à la praticabilité des routes qu'ils ont empruntées ; la technique utilisée ici consiste à accéder au champ positionHistory (A.2, p.103) de l'agent observé (ces informations sont fournies par le noyau) pour connaître les chemins empruntés par cet agent et en déduire qu'ils sont praticables[3]. Deuxièmement, comme nous l'avons vu au chapitre 4.3 (page 33), le simulateur commence par envoyer la configuration initiale du monde aux agents. Ceci concerne aussi la position initiale des autres agents, et donc des civils, ce qui permet de déduire quels sont les bâtiments susceptibles de s'être effondrés sur des civils[3]. Enfin, en ce qui concerne la classe Scope[4], elle fournit un outil de débogage visuel très utile pour suivre les actions et les décisions d'un agent en particulier.

[3] Ces points un peu irréalistes seront abordé plus en détail au chapitre 7.

[4] Nous vous invitons à découvrir la capture d'écran du *scope* au chapitre C.21, page 175.

7. Ces quatre classes concernent les agents mobiles. Nous détaillerons les engrenages de leur prise de décision au point suivant. Notons juste que la classe CivilianController n'est pas utilisée, puisque les agents civils sont fournis par le comité organisateur.
8. Les 3 dernières classes concernent les agents bâtiments. Comme les agents mobiles, ceux-ci sont présentés en détail au point suivant.

Notons encore que les classes des 4 premiers points du tableau mettent en place un système permettant de programmer à l'aide de règles de production. Celui-ci facilite la programmation des parties intelligentes du système multiagent. Par exemple, nous pouvons facilement décrire l'intelligence d'un agent qui recherche un incendie et s'y déplace grâce aux règles suivantes[5] :

$$isBurning := FIERYNESS\_F \geq 1 \wedge FIERYNESS\_F \leq 3$$
$$fireSet := \{b \in buildingSet\ in\ world \mid isBurning(b)\}$$
$$route := route\ from\ here\ to\ fireSet\ minimising\ cost\ Function$$
$$move\ along\ route$$

Le programmeur pourra aisément traduire (en java) ces règles de production de la manière suivante :

```
Predicate isBurning = FIERYNESS_F.gte(1).and(FIERYNESS_F.lte(3)) ;
Domain fireSet = world.buildingSet().get(isBurning) ;
Route route = router.get(self.position(), fireSet, costFunction) ;
move(route) ;
```

6.3 Stratégies et règles de décision de YabAI

Cette section présente les règles de stratégie que suivent les agents de l'équipe YabAI. Ces règles sont écrites dans les classes xxxController des points 7 et 8 du tableau 6.1.

6.3.1 Agents paramédicaux

Les agents paramédicaux ont un système de prise de décision relativement simple. Leur stratégie consiste en la recherche d'agents[6] blessés (enterrés) ; une fois repérés, ils tentent de les sauver ensemble en se prévenant par radio.

[5] FIERYNESS_F est une fonction qui renvoie une valeur représentant l'intensité d'un incendie. Ces valeurs vont de 0 à 7 : 0 représente l'absence d'incendie, les valeurs de 1 à 3 représentent des incendies de plus en plus importants, et les valeurs supérieures représentent un incendie éteint (naturellement ou par des pompiers).

[6] Civils ou sauveteurs, ces derniers pouvant également être blessés, brûlés, enterrés, etc.

Les agents paramédicaux classent les agents en quatre catégories : «besoin d'aide immédiate», «besoin d'aide», «pas besoin d'aide» et «abandon». Ainsi, ils peuvent optimiser leurs efforts, en sauvant prioritairement ceux qui sont gravement blessés et en ignorant ceux pour lesquels leur intervention s'avérerait inutile. Les agents paramédicaux ne disposent pas d'autre règle. Son auteur, M. Takeshi, explique que le nombre de blessés et la gravité des blessures évoluent beaucoup trop rapidement pour pouvoir optimiser un plan de sauvetage particulier.

6.3.2 Agents pompiers

La stratégie des pompiers part du constat suivant : un incendie qui se déclare peut rapidement prendre de l'envergure et devenir incontrôlable. Alors qu'un incendie naissant est facile à éteindre, un incendie de grande envergure est très difficile à maîtriser, même pour plusieurs brigades de pompiers. La priorité maximale doit donc revenir aux incendies naissants. Un autre facteur important est celui de la densité de bâtiments à proximité de l'incendie. En effet, alors qu'un bâtiment isolé ne représente pas de grand danger pour le reste de la ville, les bâtiments accolés permettent à l'incendie de se propager rapidement.

YabAI définit ainsi trois facteurs différents afin de déterminer l'incendie le plus important :

- L'intensité de l'incendie ;
- La densité de bâtiments à proximité ;
- La distance de l'agent à l'incendie.

En faisant varier ces facteurs, on peut définir une liste de priorités permettant à l'agent de déterminer l'incendie à éteindre. Les priorités telles qu'elles sont définies dans YabAI sont représentées à la figure 6.2 :

1. Incendies proches venant de se déclarer ;
2. Incendies proches de moyenne intensité ;
3. Incendies proches de forte intensité, dans un voisinage très dense ;
4. Incendies proches de forte intensité, dans un voisinage assez dense ;
5. Incendies éloignés venant de se déclarer dans un voisinage très dense ;
6. etc.

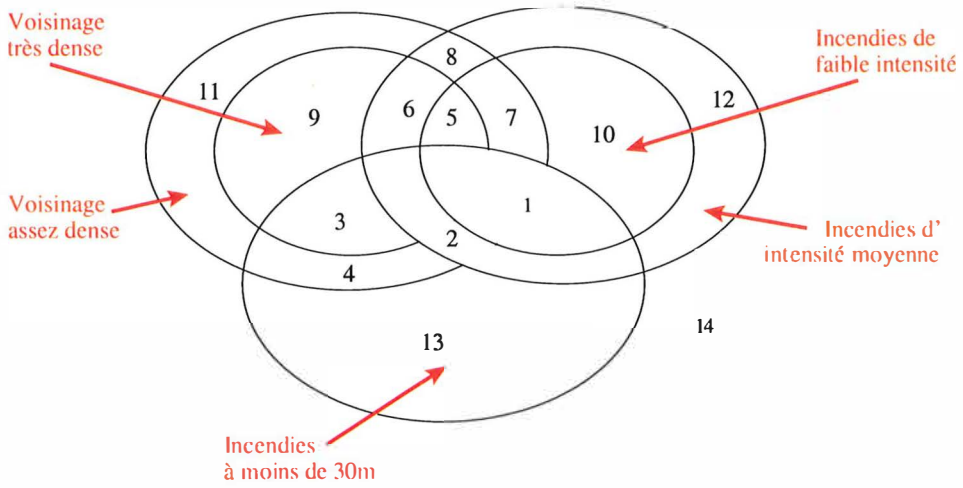


FIG. 6.2 – YabAI : Priorité d'extinction des incendies.

6.3.3 Agents policiers

Les agents policiers ont comme objectif principal le rétablissement du réseau routier. De même que les agents pompiers, ils doivent utiliser des critères leur permettant de choisir quelles routes ils doivent privilégier dans leurs déblayements. La première priorité choisie par YabAI est de faire agir les agents policiers sur demande des autres agents. Cette solution n'étant pas très rapide[7], deux autres stratégies n'utilisant pas la communication viennent s'y greffer :

La première consiste à se diriger vers la brigade de pompiers la plus proche et à déblayer les routes au passage. De cette manière, non seulement l'agent policier sait plus rapidement répondre aux demandes des pompiers puisqu'il se dirige vers eux quand il n'a pas de demande de déblayage, mais en plus, cela crée un chemin utilisable par la brigade de pompiers lorsqu'elle doit se déplacer.

La deuxième stratégie consiste à se diriger vers les incendies les plus proches et à déblayer les routes au passage. Ainsi, cela crée également un chemin possible vers les incendies pour les brigades de pompiers. De ces deux stratégies, celle qui fournit la destination la plus proche sera choisie.

[7]Comme nous l'avons vu, il faut au minimum 3 minutes pour qu'un message passe d'un agent sauveteur à un autre de type différent.

6.3.4 Agents centres

Les agents centres ne font que retransmettre l'information qu'ils reçoivent. Certaines règles sont cependant appliquées, en adéquation avec les flux de messages possibles décrits dans les points précédents :

- la caserne de pompiers retransmet les demandes de déblayage ou les signalements de blessés qui proviennent des pompiers. Les messages qui proviennent d'autres centres ne sont retransmis que s'ils viennent du commissariat et s'ils signalent des routes déblayées ;
- le commissariat retransmet les signalements de routes déblayées et de blessés qui proviennent des agents policiers. Les seuls messages en provenance d'autres centres à être répétés sont les demandes de déblayage ;
- le centre paramédical retransmet les demandes de déblaiement de ses équipes, les signalements de blessés provenant des autres centres, ainsi que les messages de signalement de routes déblayées provenant du commissariat.

6.4 Le système multiagent KADAI

Pour écrire notre système multiagent KADAI[8], nous avons donc récupéré certaines parties du code de YabAI. Nous avons en fait utilisé l'ensemble des classes des points 1 à 6 du tableau 6.1 (page 51), avec toutefois de légères modifications. La liste des modifications apportées aux classes est présentée à l'annexe D. Les autres classes des points 7 et 8 du tableau 6.1 ont été entièrement reprogrammées, avec de nouvelles stratégies de décision que nous allons maintenant exposer.

6.4.1 Règles de décision

Nous devons maintenant nous soucier de trouver des règles stratégiques afin de faire coopérer nos agents et de les rendre efficaces au sein de la simulation. L'idée initiale que nous avons explorée fut de découper la carte en quatre secteurs : Nord-Ouest, Nord-Est, Sud-Est et Sud-Ouest, et de répartir les effectifs de secours entre ces secteurs. Ainsi nous disposions deux brigades

[8]KADAI vient de Koch, ALBERT, DAMAS, et de YabAI pour les deux dernières lettres, qui proviennent sans doute elles-mêmes de *Artificial Intelligence.* Notes : ALBERT est le nom du laboratoire de recherche en intelligence artificielle (I.A.) des FUNDP de Namur, et DAMAS est le nom du laboratoire de recherche en I.A. de l'université Laval de Québec, où le stage a été réalisé. Le code KADAI est librement téléchargeable à l'adresse [Koch, 2002]

de pompiers, deux forces de police et une équipe paramédicale par secteur, les effectifs restants servant de renfort pour les situations difficiles.

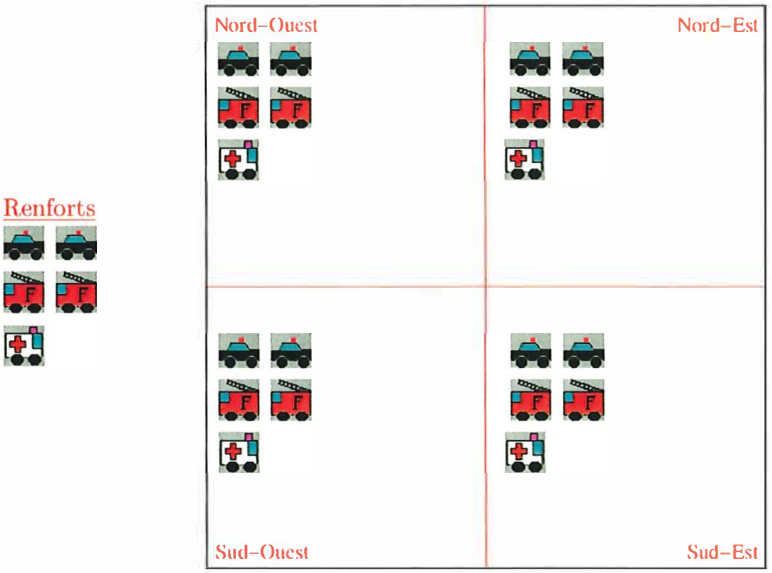


FIG. 6.3 – Division initiale des effectifs en secteurs.

Les agents sont évidemment disséminés sur toute la carte au début de la simulation. Cependant, lors de l'initialisation, chaque agent reçoit la position de tous les autres agents. Il leur est alors aisé de déterminer quel est leur secteur : les agents les plus proches de chacun des quatre coins vont au secteur respectif et les deux les plus proches du centre forment les renforts. Comme chaque agent connaît la position de ses semblables, il peut facilement déterminer s'il fait partie de ceux qui sont les plus proches d'un coin ou si le secteur respectif contient déjà son nombre d'effectifs.

Nous ne sommes malheureusement pas arrivés à créer une coopération importante entre agents d'un même secteur à cause du problème suivant, relatif au fonctionnement de la simulation. Nous aurions en effet été intéressés de voir nos agents se déplacer en groupes, les policiers déblayant devant le passage des autres. Malheureusement, la simulation fonctionne par cycles d'une minute et n'envoie pas d'information sur ce qui s'est passé dans le monde aux agents entre deux cycles. Or, la vitesse de déplacement d'un agent sauveteur

est de 960 mètres par minute[9] et son champ de vision de 10 mètres. De ce fait, nos agents ont juste le temps de voir «apparaître» les autres agents dans leur champ de vision, puis «disparaître» au cycle suivant!

Nous nous sommes alors penchés sur une approche orientée vers un seul type d'agents à la fois, plutôt que par groupe d'agents. Ceci nous a permis de revoir notre concept de secteur, de sorte qu'il corresponde mieux aux capacités des différents types d'agents. Voici les stratégies finales que nous avons décidé de mettre en oeuvre.

6.4.2 Agents policiers

Commençons par les agents policiers car ce sont ceux qui disposent de la stratégie la plus simple et, selon nous, de la plus efficace. La division en quatre secteurs ayant donné de très bons résultats, nous avons décidé de la pousser encore plus loin et d'attribuer un secteur à nettoyer pour chaque agent policier. Vous trouverez à la figure 6.4 la division en dix secteurs que nous avons utilisée.

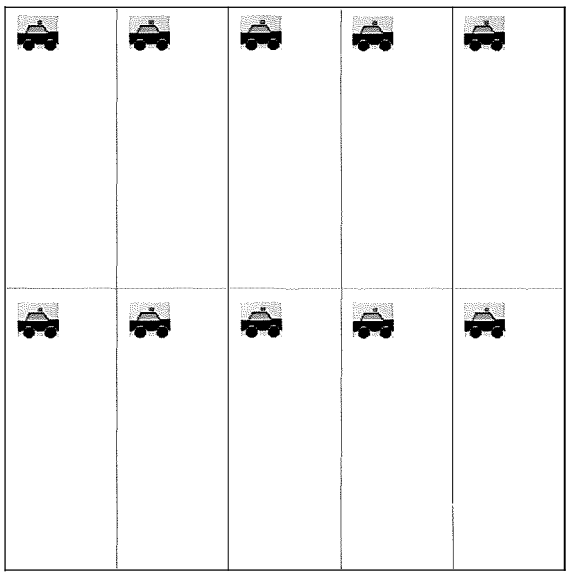


FIG. 6.4 – Répartition des policiers sur la carte.

[9]La vitesse maximale d'un véhicule étant de 16 m/s (voir les limitations au point 5.2, page 39), un agent peut parcourir jusqu'à 960 mètres par minute.

L'avantage considérable de cette technique est son extrême rapidité. En effet, chaque secteur comporte en moyenne 82 portions de route. Or, chaque portion de route prend en moyenne un seul cycle pour être nettoyée. Il existe bien sûr des endroits plus encombrés qui demandent plus de temps à être déblayés, mais d'autres ne nécessitent aucune intervention des forces de police. Si on compte une minute (un cycle) pour le déplacement d'un agent policier à la prochaine route à nettoyer et une minute pour son nettoyage, cela nous donne $2 * 82 = 164$ minutes nécessaires pour déblayer la totalité des rues de Kobe. Autrement dit, avec cette stratégie, à la moitié de la simulation[10], plus aucun agent n'a de problème de déplacement[11].

Cette technique présente d'autres avantages : non seulement les agents policiers n'effectuent jamais de déplacements inutiles car ils doivent déblayer leur secteur sans se soucier d'autre chose, mais en plus aucune communication n'est nécessaire pour que leur travail soit efficace, ce qui est très intéressant vu les difficultés qu'ont les agents pour communiquer entre eux[12].

Nous avons alors décidé d'améliorer encore cette stratégie, de manière à rendre les interactions entre les agents plus fortes. Premièrement, les agents policiers signaleront les blessés qu'ils rencontreront par radio, ce qui sera leur seule forme de communication. On peut donc les considérer comme sourds. Deuxièmement, nous avons décidé d'affecter certaines priorités concernant les routes à déblayer au sein d'un même secteur :

1. les agents policiers sont tenus de déblayer prioritairement près des bâtiments en feu ;
2. ils dégagent les routes aux positions initiales des agents. Il arrive en effet que certains agents soient complètement bloqués dès le début de la simulation. Notez que nous avons décidé de mettre cette règle en deuxième position car il nous est arrivé beaucoup plus souvent de souffrir de l'inaccessibilité d'un incendie que du manque d'un agent ;
3. s'il n'y a ni routes encombrées près d'incendies, ni agent bloqué, ils iront alors déblayer devant les bâtiments où se trouvaient initialement des civils[13], de manière à faciliter le travail des équipes paramédicales ;
4. enfin, si aucune de ces conditions n'est remplie, ils se dirigeront alors

[10]Pour rappel, on simule 300 minutes après le désastre.

[11]Nous accordons au lecteur un doute sur la possibilité que cela puisse effectivement être réellement accompli dans ces délais. Nous reviendrons sur ce sujet au chapitre 7.

[12]Pour rappel, voir point 5.4, page 40

[13]Rappelons encore une fois que le simulateur donne initialement à chaque agent la position de tous les agents du simulateur.

vers la route de leur secteur la plus proche d'eux.

Nous avons ensuite encore décidé d'améliorer ces règles de décision en incluant certaines exceptions permettant d'accélérer l'efficacité des autres agents. Nous parlons ici d'exceptions, car ces règles s'appliquent prioritairement à celles décrites ci-dessus et sont même valables hors du secteur de l'agent :

- si un agent policier aperçoit dans son champ de vision (et donc rapidement accessible) un agent sauveteur d'un autre type coincé par des débris, il ira déblayer autour de ce dernier ;
- s'il déblaye une portion d'une route qui se poursuit hors de son secteur, il continuera à en déblayer une voie de circulation[14] (nettoyage plus rapide) jusqu'au prochain carrefour. Cette règle permet d'éviter qu'un agent ne se retrouve inutilement bloqué sur une route à moitié nettoyée.

6.4.3 Agents pompiers

En ce qui concerne les agents pompiers, nous nous sommes rendu compte que la répartition par secteur n'était pas du tout efficace car les pompiers ont parfois besoin de plusieurs brigades pour arriver à se rendre maître d'un incendie.

La première règle de sélection que nous avons alors adoptée se retrouve dans celle des agents YabAI. En effet, en privilégiant l'extinction des incendies naissants, on se rend compte que les agents pompiers circonscrivent les incendies à la manière des interventions réelles. Comme vous pouvez le constater à la figure 6.5, les incendies (jaune pour naissant, orange pour intensité moyenne et rouge pour forte intensité) s'étendent sur plusieurs bâtiments. Les bâtiments de la périphérie viennent d'être incendiés et sont donc les plus faciles à éteindre (en jaune).

La deuxième règle de stratégie que nous avons développée consiste à éliminer de la sélection les bâtiments trop grands. Nous avons fixé la limite à $20m^2$ de superficie, après de nombreux tests où nos dix brigades n'ont pas suffit à éteindre l'incendie d'un unique bâtiment. On préférera alors empêcher l'incendie de s'étendre aux bâtiments environnants, plutôt que d'attaquer di-

[14] Nous n'avons jamais parlé de nettoyage partiel. En fait, en déblayant juste une voie de circulation, l'agent policier permet aux autres agents d'emprunter cette route, mais à vitesse réduite.

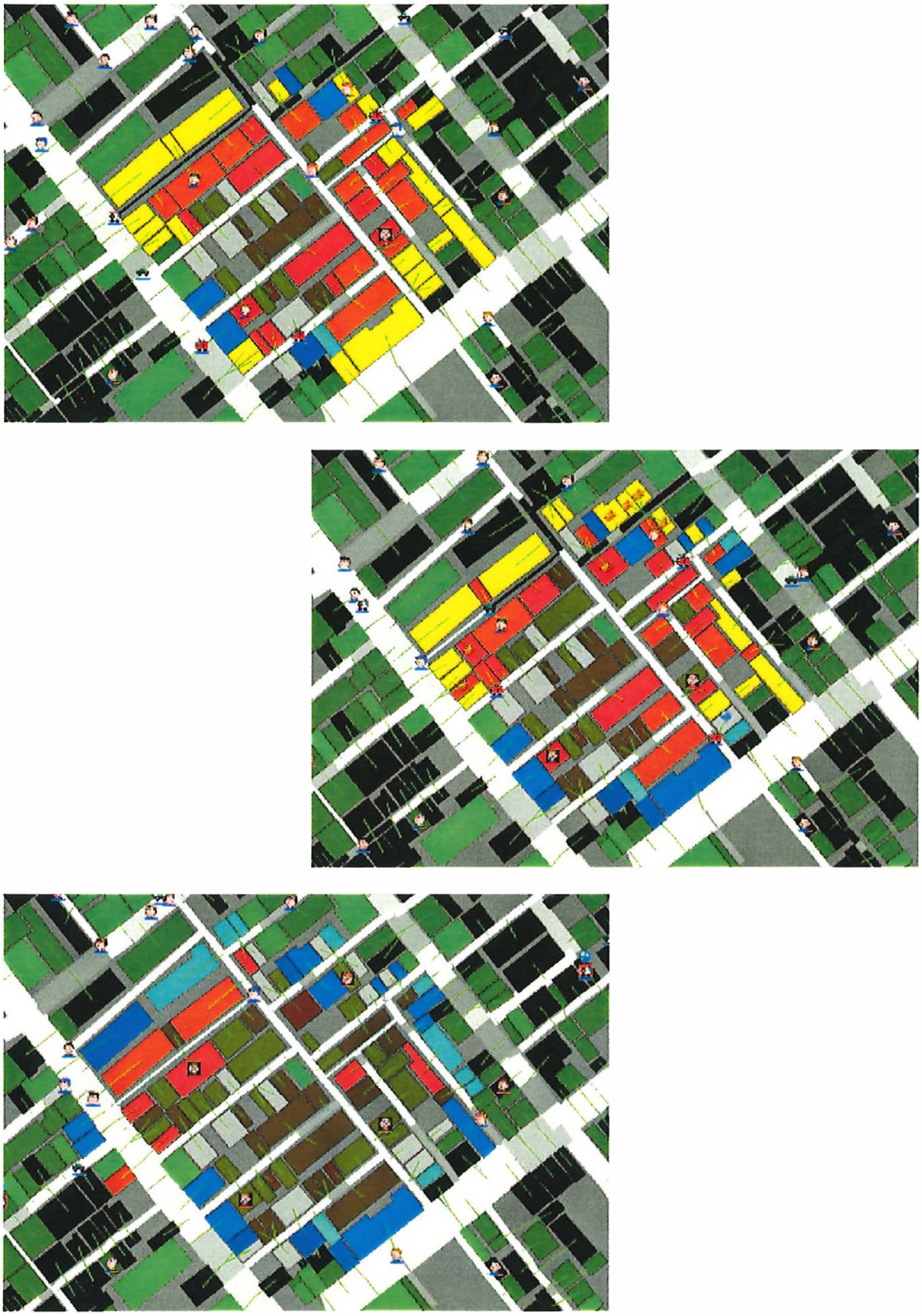


FIG. 6.5 – A la manière des intervenants réels, les agents pompiers circonscrivent les incendies.

rectement le bâtiment de plus de $20m^2$.

La troisième règle mise en oeuvre est inspirée d'une de celles des agents YabAI. Il s'agit d'éviter qu'un bâtiment ne brûle à proximité d'autres et ne permette à l'incendie de prendre des proportions importantes. Nos agents repèrent donc les incendies isolés sur un seul bâtiment, vérifient si le bâtiment présente un risque pour son entourage et, si c'est le cas, tentent de l'éteindre rapidement même si l'incendie est déjà d'intensité importante. Cette analyse ne se fait cependant que lorsqu'il n'existe plus d'incendie de faible intensité, pour éviter de perturber le travail principal des pompiers (consistant à circonscrire les incendies).

Enfin, si plus aucun bâtiment ne correspond aux critères de sélection précédents, les pompiers s'occupent des bâtiments plus proches d'eux-même. S'il n'existe plus aucun incendie, les pompiers patrouillent alors à la recherche de blessés. Comme les policiers, si les pompiers rencontrent des blessés, ils les signalent par radio aux équipes paramédicales.

Nous avons développé, outre ces règles de sélection, des règles de synchronisation afin de faire travailler coopérativement les agents pompiers. Nous avons découvert empiriquement le temps moyen nécessaire à l'extinction d'un petit bâtiment en fonction du nombre de brigades qui l'arrosaient. Voici nos résultats :

- 2 brigades : de 3 à 4 minutes ;
- 5 brigades : généralement 1 minute ;
- 10 brigades : 1 minute.

Le temps d'extinction pour un petit bâtiment nous semble trop long pour deux brigades. En effet, sur 3 à 4 minutes, l'incendie s'est généralement propagé au voisinage, et nos agents travaillent comme s'ils «couraient» après l'incendie sans jamais le rattraper. Par contre, il n'est pas nécessaire de les faire travailler à dix, puisqu'ils arrivent généralement à éteindre un petit bâtiment en une minute à cinq brigades. C'est pourquoi nous avons choisi de les faire travailler en deux escouades de cinq brigades.

Cependant, toutes les situations ne nécessitent pas qu'ils travaillent ensemble. Leur travail individuel est même préférable dans certaines conditions. Aussi avons-nous mis en place le système suivant :

T=0 durant la première demi-heure, nous considérons qu'il n'est pas nécessaire de regrouper les pompiers. Non seulement parce que les routes sont très encombrées et qu'ils passeraient plus de temps à chercher un chemin valide qu'à faire leur travail, mais surtout parce que les incendies sont très jeunes et que les pompiers, disséminés sur la carte, sont capables de maîtriser ces incendies seuls ;

T=30 après la première demi-heure, nous faisons une première synchronisation. Celle-ci ne concerne que les brigades qui ne sont pas en train d'éteindre un incendie. En effet, nous ne désirons pas gêner le travail des pompiers en train de terminer l'extinction d'un incendie, ou d'en empêcher l'expansion. Pour ceux qui ne font rien, il est fort probable qu'ils soient toujours en train de chercher un chemin pour atteindre leur cible. C'est pourquoi nous décidons de les synchroniser sur le bâtiment incendié le plus proche du centre. D'une part, nous avons choisi ce point car c'est l'endroit en moyenne le plus proche pour tous les agents et, d'autre part, nous y amenons plusieurs pompiers car il devient nécessaire, après une demi-heure d'incendie, de commencer à s'organiser pour en venir à bout.

T=50 à la cinquantième minute, ce sont tous les agents qui sont appelés à se réunir en deux groupes (en fonction de leur identifiant) sur les deux bâtiments incendiés les plus proches du centre. Nous n'avons plus peur de perturber le travail des pompiers qui seraient en train d'éteindre un incendie, puisque soit ces pompiers sont ceux qui se sont déjà dirigés au centre à la trentième minute et qui n'en sont vraisemblablement pas très loin, soit ce sont ceux qui sont en train de tenter d'éteindre un incendie seul, ce qui ne doit plus être très efficace vu les proportions qu'un incendie peut prendre en cinquante minutes.

Outre cette technique de synchronisation, notons aussi l'existence d'un mécanisme qui permet aux pompiers qui ne trouvent pas de chemin vers leur point de synchronisation de prendre seul leurs initiatives. S'ils ne sont pas efficaces, au moins pourront-ils empêcher une progression trop rapide de l'incendie, le temps que les équipes synchronisées arrivent et les intègrent à leur groupe.

6.4.4 Agents paramédicaux

Nous avons décidé de garder notre division en quatre secteurs pour les équipes paramédicales. Nous nommerons l'équipe sans secteur l'«équipe volante». Globalement, les équipes paramédicales vont creuser les tas de gravats

pour secourir les civils enfouis sous les décombres, et l'équipe volante va se charger de secourir les blessés signalés par radio. Lorsqu'il n'y en a pas, elle aidera les autres équipes, dégageant les civils du bâtiment le plus proche de sa position.

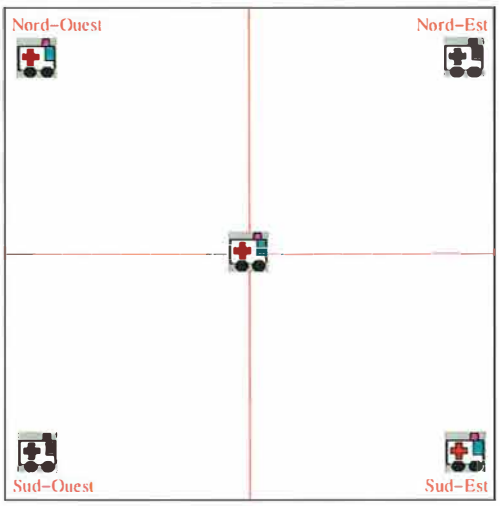


FIG. 6.6 – Répartition des équipes paramédicales sur la carte.

En ce qui concerne les équipes réparties dans les secteurs, voici leurs règles de décision. Chaque équipe commence par se diriger, dans son secteur, vers le bâtiment le plus proche susceptible de contenir des civils[15]. Une fois sur place, soit le bâtiment n'a pas été trop gravement endommagé et les civils ont pu s'enfuir, auquel cas l'équipe paramédicale passe au bâtiment suivant, soit des civils sont à sauver dans le bâtiment. L'équipe paramédicale va alors évaluer le temps qui lui sera nécessaire pour dégager les civils des décombres.

Pour ce faire, les agents ont accès à la propriété *buriedness*[16] qui donne en minutes le temps nécessaire à une équipe paramédicale pour dégager le civil des décombres.

Nous avons déterminé empiriquement que ces valeurs sont toujours initialisées à 20, 30 ou 60 minutes. Nous avons donc établi nos règles en fonctions des valeurs de cette propriété :

[15] Rappelons encore une fois que le simulateur envoie à chaque agent la position initiale de tous les agents (sauveteurs et civils) à l'initialisation de la simulation.

[16] Voir les propriétés des agents civils au point A.2, p.102. Nous reparlerons de la disponibilité d'une telle information pour les agents sauveteurs au chapitre **7**.

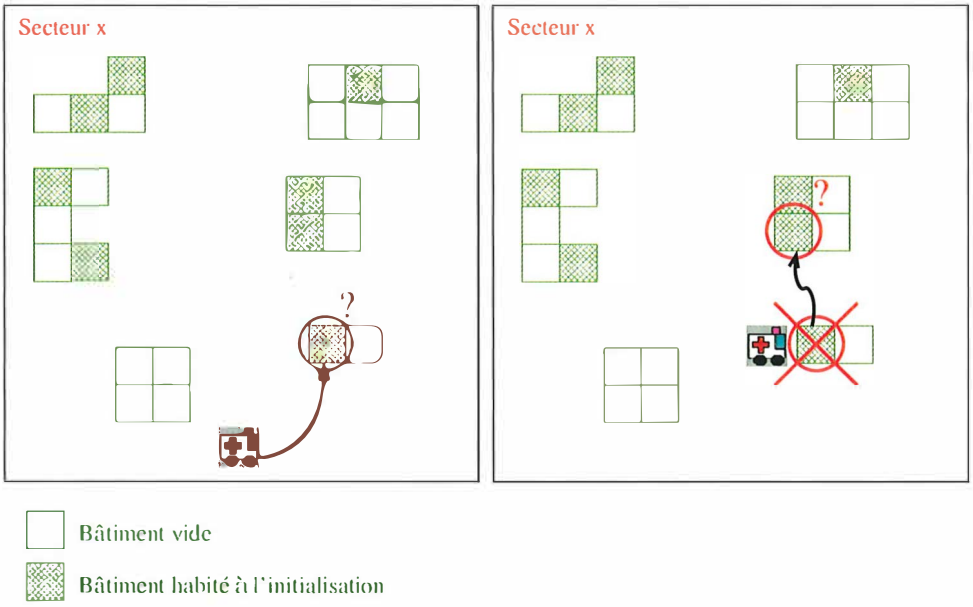


FIG. 6.7 – Déplacement des équipes paramédicales.

- moins de 20 minutes (y compris) : sauver le civil ;
- plus de 30 minutes (non compris) : postposer le sauvetage du civil ;
- entre 20 et 30 minutes : cela dépend de l'état des routes. Si l'équipe sait se déplacer rapidement pour aller trouver des civils plus aisés à sauver, il sera plus opportun de postposer le sauvetage du civil. Sinon, il vaudra mieux le dégager plutôt que de perdre son temps à trouver un chemin utilisable. Pour déterminer quand les routes sont utilisables, nous nous sommes basés sur la rapidité des forces de police à nettoyer les routes. Nous savons qu'il faut en moyenne 164 minutes pour tout nettoyer. Nous en avons déduit qu'à la moitié de cette valeur, les routes principales devaient être dégagées et permettre aux équipes paramédicales d'aller voir d'autres bâtiments. Nous avons fixé empiriquement cette valeur à une heure et quart (75 min).

Une fois que tous les civils d'un secteur sauvables en moins de 20 (ou 30) minutes sont sauvés, l'équipe paramédicale change de secteur. Ce changement se fait dans le sens des aiguilles d'une montre[17]. Une fois tous les secteurs

[17]Le sens de ce changement est arbitraire. Le seul point important est de faire suivre le même sens à tous les agents, afin d'éviter des changements chaotiques qui pourraient s'avérer inefficaces.

visités, l'équipe paramédicale s'occupera des civils dont elle a postposé le sauvetage.

6.4.5 Agents centres

Les agents centres ne font que relayer l'information. Nous avons récupéré le système de communication de YabAI et nous l'avons sensiblement simplifié. En effet, nous n'avons plus qu'un seul type de message qui circule, à savoir celui qui sert à signaler aux équipes paramédicales les blessés rencontrés par les différents agents. Le flux de messages n'a plus que deux origines et une seule destination : des pompiers ou des policiers aux équipes paramédicales. Il n'y a donc plus énormément de possibilités de goulot d'étranglement. Les pompiers et les policiers sont sourds aux communications et les équipes paramédicales sont sourdes et muettes. La seule exception est l'équipe paramédicale volante, qui ne reçoit au maximum que les quatre messages que peut envoyer son centre. Les seules pertes de messages qui peuvent alors arriver se situent au niveau des centres.

Le commissariat et la caserne de pompiers n'écoutent que les messages provenant des agents qui en dépendent, ce qui peut faire tout de même jusqu'à 40 messages. Nous aurions pu limiter ce chiffre à dix en envoyant toutes les informations que veut communiquer l'agent dans un seul message plutôt que dans quatre étant donné l'absence de réglementation des messages échangés, mais nous avons préféré suivre la structure des messages de YabAI, à savoir un message par objet[18]. Au niveau du centre paramédical, ne sont écoutés que les messages des deux autres centres, ce qui réduit le nombre possible de réceptions à huit messages par minute. Nous aurions pu limiter le nombre d'émissions de messages des deux autres centres à deux, pour être sûr qu'il n'y ait jamais surcharge au niveau du centre paramédical. Cependant, le nombre de messages effectivement échangés n'est pas très important et les agents sauveteurs répètent chaque message autant de fois qu'ils aperçoivent le blessé concerné, ce qui rend la perte de message moins problématique.

[18]La classe Message de YabAI (C.16, p. 160) est prévue pour envoyer une liste d'ID (entiers identifiants des objets), avec un prédicat codant la signification du message, mais dans les classes de contrôle, seuls des messages de 2 entiers (un prédicat et un ID) sont échangés.

6.5 Résultats

Nous allons présenter ici les résultats des tests comparatifs que nous avons effectués sur le simulateur. Nous avons testé nos agents et ceux de YabAI sur cinq situations initiales différentes. Comme le système n'est pas déterministe, nous avons lancé plusieurs tests par situation initiale. Vous trouverez le détail des informations collectées à l'annexe E[19].

Nous avons reporté les moyennes de ces résultats sur les quatre graphiques de la figure 6.8. L'ordonnée des graphiques est toujours le numéro de la situation choisie. Le premier de ces graphiques montre les résultats globaux de l'évaluation de nos agents. Rappelons que l'évaluation est une fonction à minimiser qui représente les pertes humaines et matérielles. Nous pouvons facilement voir que nos agents obtiennent toujours une évaluation inférieure à celles des agents YabAI, ce qui signifie qu'au départ de ces cinq situations, nos agents sont plus aptes à faire face au désastre. On peut remarquer aussi que les deux courbes évoluent de la même manière, ce qui est significatif de la difficulté de la situation choisie pour chaque carte. Par exemple, la troisième carte est celle dont le nombre d'incendies est le plus élevé. De plus, ils sont isolés dans un coin de la carte alors que les pompiers sont isolés dans le coin opposé. Sur une telle carte, il faut alors commencer par déblayer un chemin en diagonale avant que les pompiers ne puissent enfin s'occuper des incendies.

Le deuxième graphique représente le nombre de morts qu'il y a eu au cours de la simulation. On peut remarquer que ce graphique est fort identique au précédent, ce qui découle directement du calcul de la fonction d'évaluation. En effet, rappelons que les pertes matérielles et physiques[20] représentent la partie décimale de l'évaluation, alors que les pertes humaines en forment la partie entière.

Le troisième graphique nous montre la surface totale incendiée. A première vue, il semble difficile de pouvoir tirer une conclusion. Si on observe les situations 1, 4 et 5, la différence est importante, nos agents préservant plus de

[19] Le lecteur pourrait s'étonner de ne pas voir plus de situations testées. La raison découle du temps nécessaire à la réalisation de ces simulations. Pour rappel, une simulation dure une vingtaine de minutes. Considérant que le pourcentage de *crashs* du noyau et autres bogues des autres modules représente approximativement 50% du total des simulations effectuées, on peut considérer que chaque simulation rapportée à l'annexe E nous prit 40 minutes. Notons aussi qu'il n'a pas été possible d'automatiser ces tests. Au total, il s'agit de 137 tests réalisés, ce qui correspond à une durée d'environ 3 jours, 19 heures et 20 minutes sans interruption.

[20] Entendez ici les blessures, les brûlures, etc.

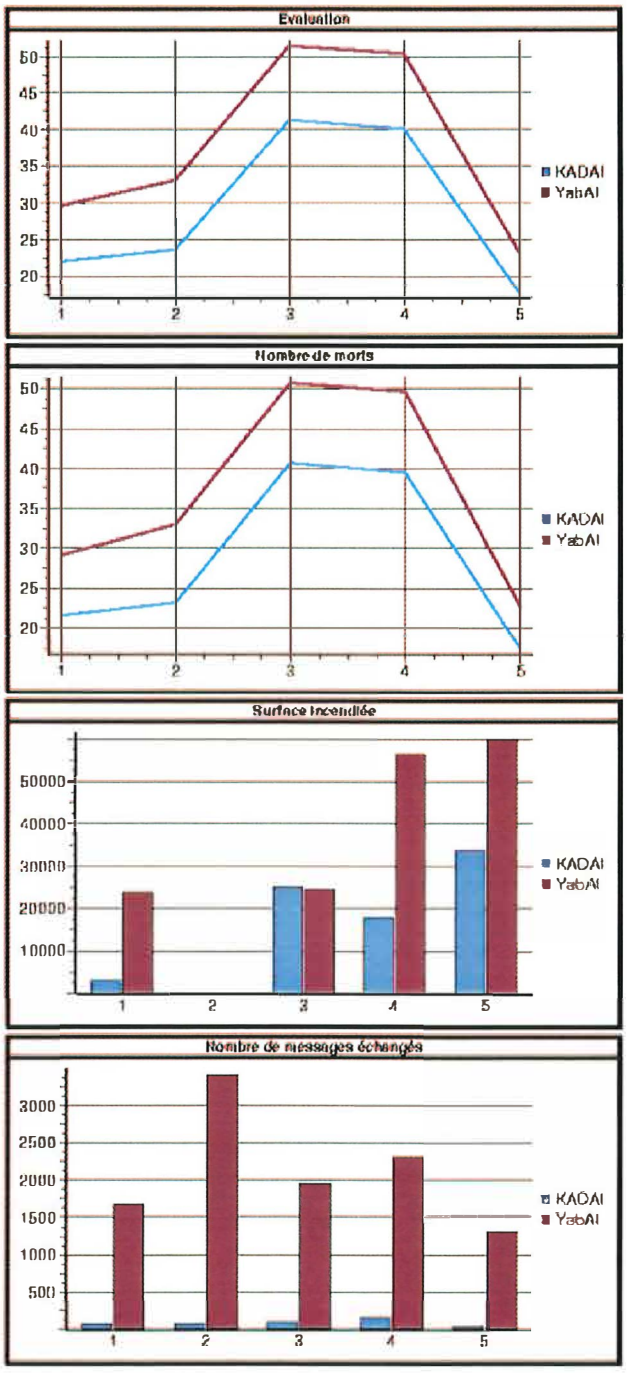


FIG. 6.8 – Comparaisons entre agents YabAI et KADAI

la moitié des bâtiments incendiés dans les simulations de YabAI. Cependant, la situation 3 présente nos agents comme moins performants. La différence étant minime, nous pourrions tout de même estimer que nos pompiers sont meilleurs dans l'ensemble. Sans doute serait-il nécessaire d'effectuer d'autres tests sur d'autres situations initiales pour vraiment pouvoir conclure avec plus de certitude.

Le dernier graphique, quant à lui, nous montre une différence flagrante entre nos agents et ceux de YabAI. En effet, nous émettons entre 2 et 7% du nombre total de messages que s'échangent les agents YabAI. Sans doute existe-t-il là une possibilité d'amélioration de nos stratégies de décision, en profitant un peu plus de la communication.

Chapitre 7

Critiques de la simulation

Vous avez sans doute souvent aperçu, tout au long de votre lecture, des notes de bas de page renvoyant à ce chapitre, lorsque certains faits pouvaient prêter à discussion quant à leur réalisme. Le but de ce chapitre est d'aborder tous ces problèmes, et même plus, car vous y découvrirez aussi tous les problèmes techniques qui rendent la réalisation d'un système multiagent pour le concours de 2001 assez délicate.

Mais gardons aussi à l'esprit que ce concours est le premier d'une longue série et devait vraisemblablement passer par des erreurs de jeunesse. Simuler autant de faits réels n'est pas une chose aisée, d'autant que de nombreuses disciplines (autres que celle de l'informatique) se croisent à ce niveau, mélangeant la psychologie cognitive (notamment le comportement des foules en panique), la physique (étude de l'évolution des incendies, possibilités de communications en pareille situation...), la médecine (évolution des blessures graves et des grands brûlés), l'étude de stratégies d'interventions en cas de catastrophe naturelle, et la liste est encore longue. Il faudra bon nombre de spécialistes pour arriver à mettre tout le monde d'accord sur chacun des paramètres qui peuvent entrer en compte dans cette simulation.

Finalement, nous voulions aussi signaler que les critiques qui sont émises ici le sont à notre niveau de connaissance et, n'étant pas spécialiste dans chacun des domaines concernés, notre avis peut ne pas toujours correspondre à la réalité. Cependant, la plupart des points suivants ont déjà été abordés par le comité, parfois même après une intervention de notre part, ce qui devrait permettre une certaine crédibilité de nos propos (Voir [Proposal, 2001])

7.1 Le réalisme

7.1.1 Les informations surréalistes

Le premier point que nous désirons soulever concerne les informations auxquelles ont accès nos agents sauveteurs. Il s'agit de la position initiale des agents, de l'accès aux champs *buriedness, positionHistory*, et de l'information visuelle concernant les incendies.

Agents extralucides

Commençons par l'omniscience initiale des agents. Nous pensons qu'il est tout à fait anormal que les agents connaissent la position exacte de chacun des autres agents. Cela reviendrait alors à penser que chaque civil (et chaque équipe de sauvetage) se déplacerait avec un émetteur signalant exactement sa présence, ce qui ne nous semble pas (actuellement) réaliste. Peut-être pourrait-on déterminer certaines probabilités quant à la présence de civils dans certains bâtiments fonctionnels (par exemple en fonction des heures de bureau, des types de bâtiments, etc.), mais certainement pas pour tous les bâtiments, ni avec certitude.

Nous pourrions éventuellement accepter que les agents de même type connaissent leurs positions, voire celles de tous les agents sauveteurs, en supposant qu'ils disposent de systèmes semblables aux GPS qui leur communiqueraient la position de leurs collègues. Cependant, si telle était la réalité, nous pensons que cette information devrait rester disponible au cours de la simulation, du moins tant que l'infrastructure nécessaire à œ système reste valide.

Le comité organisateur a repéré ce problème (certaines équipes s'étant servie de ces informations) et l'a par ailleurs qualifié de «*unbelievably unusual*[1]». Il a dès lors fait savoir que ces informations ne seraient plus disponibles lors des futures compétitions[2].

Agents sourciers

En ce qui concerne l'accès au champ *buriedness*, les agents, et en particulier les équipes paramédicales, peuvent, d'un simple coup d'oeil, déterminer le temps exact qui leur sera nécessaire pour déterrer les civils. Nous estimons

[1] Incroyablement inhabituel.
[2] Voir [Proposal, 2001], proposition 1, point VI.(2).

que cette information devrait être beaucoup plus floue, de manière à refléter une appréciation plutôt qu'une certitude.

Pour illustrer notre propos, remémorons-nous la tragique attaque terroriste des tours du World Trade Center, et particulièrement la difficulté qu'ont eu les secours pour localiser les civils enfouis sous les décombres. Nous sommes évidemment bien conscient que chaque bâtiment de Kobe n'a pas l'ampleur de ces tours américaines, mais nous pensons pouvoir affirmer qu'il existe de nombreux bâtiments importants pour lesquels un simple coup d'oeil ne suffirait pas à déterminer le temps exact nécessaire au sauvetage de civils qui en seraient prisonniers[3].

Un autre point irréaliste de ce champ concerne les valeurs que lui affecte le simulateur. En effet, tout agent, avant intervention des équipes paramédicales, a la valeur de son champ *buriedness* fixée à 0, 20, 30 ou 60 minutes. Bien que ceci provienne vraisemblablement d'une simplification volontaire de la part des réalisateurs du simulateur, nous nous interrogeons, d'une part, sur l'absence de commentaire ou de prévision d'amélioration de la part du comité à ce sujet et, d'autre part, sur la durée maximale qui semble avoir été évaluée aux environs d'une seule heure. Nous pensons que la réalité présente plus que probablement des situations nécessitant bien plus de temps, notamment lors d'effondrement de bâtiments de taille importante.

Pour résumer ce point, nous pensons que cette information devrait, à l'avenir, s'avérer beaucoup plus floue, que le temps effectif maximal pour un sauvetage devrait devenir plus important et, pourquoi pas, pouvoir dépasser le temps de la simulation. Cette situation ne nous semblerait pas irréaliste, dans l'hypothèse selon laquelle l'agent pourrait sous-évaluer un sauvetage qui se présenterait beaucoup plus délicat que prévu.

Agents historiens

L'accès au champ *positionHistory* donne la possibilité aux agents de connaître l'ensemble des chemins empruntés par les autres agents dès qu'ils entrent dans leur champ de vision. Cette information est certainement trop complète, d'autant plus qu'elle se transmet automatiquement avec les informations visuelles envoyées par le noyau.

[3]Nous invitons le lecteur à reconsulter les images 3.4, 3.5 et 3.6 (p. 23) de l'après tremblement de terre de Kobe pour qu'il juge par lui-même.

Considérons par exemple que nous voyons une personne arriver d'une rue perpendiculaire à la nôtre. Nous pourrions probablement supposer qu'au moins une partie de cette rue est praticable. Mais peu de choses peuvent nous faire savoir d'où vient exactement cette personne, et encore moins quel chemin (sachant que peu de routes sont valides) elle a emprunté pour parvenir jusqu'à nous.

Comme suggestion à ce sujet, nous proposerions de rendre accessible uniquement l'information concernant le chemin emprunté dans le champ de vision de l'agent qui en voit arriver un autre. En effet, ce sont des informations visuelles auxquelles on peut avoir accès dans la réalité, mais qui sont rendues inaccessibles par le fonctionnement par minute du simulateur[4].

Le comité s'est penché sur la question[5] et a décidé de remplacer le contenu du champ *positionHistory* par le chemin le long duquel s'est déplacé l'agent lors du dernier cycle (il sera vide s'il ne s'est pas déplacé), et non son déplacement total. De plus, l'accès à ce champ sera désormais interdit. Le comité semble conscient de la controverse que peut susciter cette interdiction, mais pense que le noyau devrait réaliser un travail trop important en coupant la partie non visible de chaque agent.

Agents à vision infrarouge

La dernière information à laquelle les agents ont accès et qui, à notre sens, ne reflète pas la réalité concerne les incendies. La décision de permettre aux agents de savoir si un bâtiment est incendié, et ce à n'importe quelle distance, vient sans doute de la maxime «il n'y a pas de fumée sans feu». Il nous parait cohérent que, lorsqu'un important incendie fait rage, nous soyons en mesure de le voir (ou d'en voir la fumée) dans un rayon de 700m[6] (et bien plus !).

Par contre, ce qui nous paraît moins certain, c'est qu'il soit possible de voir, de n'importe quel point de la ville, les incendies qui viennent de s'y déclarer. Il nous paraît cependant difficile de pouvoir faire une suggestion précise, car nous ne disposons pas d'assez d'informations concernant les 3 valeurs d'intensité possible pour un incendie[7]. Par exemple, que signifie un

[4]Nous y reviendrons dans les problèmes techniques au point 7.2.1.

[5]Voir [Proposal, 2001], proposition 1, points I.(11) et VI.(3).

[6]La simulation couvre une région de 500m × 500m. La diagonale, distance maximale en ligne droite, mesure alors $\sqrt{500^2 + 500^2} \simeq 707m$.

[7]Le champ *fieryness* des bâtiments est défini à l'annexe A.11, page 108.

bâtiment ayant son champ *fieryness* à 2? A partir de quelle distance cet incendie de «moyenne intensité» peut-il se voir... ?

Le comité a réagi à cette problématique, non en précisant les valeurs du champ *fieryness* mais en posant la règle suivante : tout incendie dont la valeur *fieryness* est égale à 1 prendra deux ou trois cycles avant d'être découvert à plus de 100 mètres. Nous pensons que même si cette mesure correspond à une amélioration du réalisme de la simulation, une étude plus rigoureuse devrait être menée[8] pour déterminer quand un incendie est visible à distance. Parallèlement, les valeurs du champ *fieryness* devraient être définies plus clairement.

7.1.2 Les handicaps des agents surspécialisés

Notre seconde interrogation concerne ce que nous appelons la surspécialisation des agents. Nous nous sommes fait cette réflexion lorsque nous avons réalisé que nos agents policiers n'étaient plus très utiles une fois leur travail de déblayage terminé. Il en est de même pour nos agents pompiers lorsqu'ils arrivent à maîtriser tous les incendies avant la fin de la simulation. La seule utilité que nous ayons pu leur trouver fut de les faire patrouiller pour renseigner les équipes paramédicales sur la position des civils blessés. Nous pensons que dans pareille situation, les postes ne sont pas immuables.

En effet, même si elle ne dispose pas du bon matériel, une équipe de policiers (ou de pompiers) peut très bien s'improviser ambulance pour conduire un blessé à l'hôpital ou encore dégager un civil coincé sous quelques gravats de taille raisonnable. Nous irions même jusqu'à penser qu'une partie des civils eux-mêmes tenteraient d'aider partout où ils peuvent.

C'est en tout cas ce que semblaient penser les développeurs d'agents civils lorsqu'ils ont présenté de nouvelles règles comportementales[9]. L'une de celles-ci proposait que les agents civils aident volontairement les agents sauveteurs en leur signalant la position des agents ensevelis. Nous pensons que la coopération devrait aller plus loin, en permettant à tous les agents de creuser ou de déblayer des gravats, à rapidité dépendante de leur spécialisation.

[8] Peut-être une telle étude a-t-elle été menée, auquel cas nous sommes dans l'erreur. Cependant, nous n'en avons pas trouvé de trace, ni de définitions précises des valeurs du champ *fieryness*, ce qui nous a amené à penser que ceci restait sans doute encore assez informel.

[9] Voir [Proposal, 2001], proposition 7.

7.1.3 La communication limitée

Nous nous sommes interrogés sur la pertinence de la limite de 4 messages pour les centres, qui devraient, selon nous, être capables de gérer plus de messages qu'un simple agent, et cela même s'ils sont partiellement endommagés. Nous supposons en effet que pareils bâtiments doivent posséder un central composé de plusieurs relais de communication et que, vu la situation, il devrait s'y trouver plus d'une personne pour organiser les secours.

Après en avoir discuté avec le comité[10], celui-ci a pris la décision d'augmenter la limite du nombre de messages audibles par les centres et non celle du nombre de messages pouvant être émis. La capacité d'audition est passée de 4 à 10 messages pour le centre paramédical et à 20 messages pour les autres.

Nous pensons que cette augmentation est une bonne chose, bien que nous ne connaissions pas la réalité des moyens de communication de ces centres. Nous nous posons en outre encore la question de savoir si les communications inter-centre ne devraient pas suivre d'autres voies de communication et être masquées aux agents sur le terrain.

Un autre point, sans doute plus important encore que la définition de la limite de message, concerne l'absence de réglementation des messages échangés. Sans limite du nombre de caractères, ni du nombre d'informations pouvant être échangées, il est tout à fait possible que deux agents s'échangent l'entièreté de leur base de connaissance, ce qui nous semble tout à fait irréaliste en moins d'une minute.

Après avoir discuté de ce problème avec le comité[11], celui-ci a proposé une ébauche de réglementation. Chaque message devra être composé, outre de l'identifiant de l'émetteur du message et de la taille de ce message, d'un code sémantique déterminant le type de message[12] et de l'identifiant de l'objet concerné par ce message.

Nous pensons que cette réglementation n'est pas assez souple car, bien qu'elle empêche deux agents de s'échanger l'ensemble de leur savoir, elle interdit un message tel que «rues A, B et C déblayées» qui, à notre sens, devrait être possible. Nous suggérerions donc d'accepter un certain nombre

[10] Voir [Proposal, 2001], propositions 3 et 4.

[11] Idem.

[12] Ce code peut signifier par exemple «blessé ici», ou encore «route nettoyée», etc.

d'identifiants après le code sémantique. Nous proposerions de le fixer entre 3 et 5.

7.1.4 Problèmes spécifiques aux pompiers et aux policiers

Le premier problème que nous rapportons ici concerne les pompiers. Le fait qu'ils puissent avoir accès à une quantité illimitée d'eau et que cette eau provienne directement de leur camion n'est pas réaliste. Cependant, il s'agit ici d'une volonté de simplification de la part du comité, puisqu'ils mentionnent dans leurs manuels qu'à partir des versions 1.x du simulateur, il n'en sera plus ainsi. En effet, les pompiers devraient d'abord trouver des bouches d'eau valides, chacune ayant des débits différents en fonction de l'état d'endommagement des canalisations, avant de pouvoir s'y connecter et de commencer à éteindre les incendies. La compétition de 2002 sera déjà améliorée par la possibilité d'arroser dans deux directions à partir d'un camion, en divisant le débit en deux pour chaque source[13].

Le second problème concerne la rapidité des policiers à déblayer les routes. Rétablir l'ensemble du réseau routier en moins de trois heures est une perspective très optimiste. Lorsque nous revoyons par exemple l'image de ce bâtiment entièrement effondré sur la route[14], nous pensons que plusieurs heures seront bien nécessaires aux forces de police pour rendre cette voie utilisable. Or, dans l'état actuel de la simulation (simulateur 0.31), la majorité des routes sont déblayables en moins de 3 minutes, les plus longues ne dépassant jamais les dix minutes.

Le comité semble s'en être rendu compte, car dans une de ses propositions[15] concernant le prochain championnat, il parle d'une vitesse moyenne de déblayage des routes de $20m^2$ par minute qui sera maintenue au prochain championnat pour des raisons de compatibilité, mais qui lui semble toujours être trop courte. Il est donc fort probable que la compétition de 2003 voie une baisse d'efficacité des agents policiers.

[13] Voir [Proposal, 2001], proposition 1, point III.(1).
[14] Nous parlons de l'image 3.4 à la page 23.
[15] Voir [Proposal, 2001], proposition 1, point II.(1).

7.2 Problèmes techniques

Le second point de ce chapitre concerne les problèmes liés à la conception et à la réalisation des différents modules. Il ne s'agit plus du problème concernant la manière dont on désire représenter le réel, mais des problèmes qui n'existent que par l'utilisation de l'outil informatique pour représenter ce réel. On y discutera des informations et des possibilités que l'on perd en simulant le temps de manière discrète ; des problèmes liés au critère de décision concernant la vision ou non d'un objet, et principalement de certaines routes ; de la manière dont sont gérés les déplacements ; de la possibilité qu'ont parfois les civils de s'échapper des décombres ; des destins liés entre eux ; et encore d'autres problèmes.

7.2.1 L'agent invisible

Le premier problème technique auquel nous avons été confronté concerne la simulation du temps. Le pas d'incrémentation d'une minute nous semble beaucoup trop grand par rapport à la vitesse à laquelle se déplacent les agents. Ceci peut devenir gênant non seulement lorsque deux agents se croisent, mais aussi lorsqu'ils désirent se suivre.

Rappelons le problème : les agents se déplacent à une vitesse de 16 m/s, et donc parcourent, en un cycle d'une minute, 960 mètres. Or, chaque agent ne reçoit du noyau les informations sur ce qu'il voit qu'une seule fois par cycle, à un moment bien déterminé. Ainsi, le noyau n'envoie aucune information sur ce qui s'est passé entre deux de ces cycles, c'est-à-dire entre deux minutes. Tout se passe alors pour l'agent comme s'il voyait une photographie de la réalité simulée chaque minute, sans en saisir les mouvements.

Prenons pour exemple la figure 7.1 où un agent policier A immobile désire interpeller un agent pompier B mobile[16]. Considérons que l'intérieur du cercle représente le champ de vision de A et, pour simplifier, son champ auditif. Nous avons aussi représenté un agent pompier B sur le petit point noir en haut à droite, au temps T=0. A ce moment, A ne peut donc pas le voir et ignore où il se trouve. Supposons alors que B se déplace le long de la ligne pour venir se placer sur le petit point noir du centre, dans le champ de vision de A, au temps T=1. A ce moment précis, le noyau envoie les informations à A qui va alors voir «apparaître» B dans son champ de vision, et ce, sans

[16]Notons que le même genre de raisonnement peut s'appliquer pour deux agents qui désireraient se suivre.

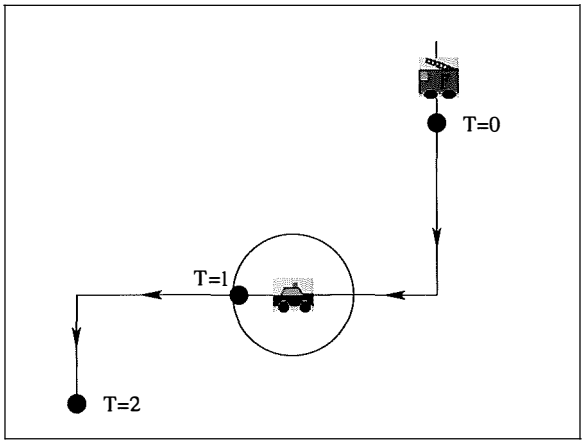


FIG. 7.1 – Un agent passant dans le champ de vision d'un autre.

savoir d'où il est arrivé[17]. Notons également que si B était arrivé un peu plus loin, hors du champ de vision de A, A n'aurait même pas eu connaissance du passage de B dans son champ de vision.

Revenons au cas où B se trouve dans le champ de visions de A (T=1). Supposons que A désire interpeller B. Il envoie sa communication au noyau, qui va la faire arriver, au cycle suivant (T=2), à tout agent étant dans le champ auditif de A. Or, si B a continué son déplacement (point noir en dessous à gauche), il ne sera plus dans le champ de vision de A et n'entendra alors pas son interpellation.

Bien que nous pensons que le pas d'incrémentation du temps soit trop important, il n'est sans doute pas encore techniquement souhaitable de le diminuer jusqu'à ce qu'il soit suffisamment petit pour régler ce problème. De fait, si nous le réduisions à une seconde, nous alourdirions considérablement la simulation sans arriver à régler complètement les problèmes qui y sont liés. En effet, une seconde de cycle permettrait aux agents de parcourir 16 mètres, ce qui est toujours suffisant pour traverser le champ de vision d'un autre agent sans être vu. Un dixième de seconde serait sans doute l'unité souhaitable, mais rendrait probablement la simulation inutilisable[18]. Une autre solution sans doute moins «coûteuse» serait d'envoyer des informations concernant

[17]Sauf si A interroge le champ *positionHistory* de B, ce qui sera plus que probablement interdit à l'avenir.

[18]Rappelons qu'actuellement, une simulation dure déjà environ 20 minutes. Utiliser un pas d'incrémentation du temps d'un dixième de seconde ferait vraisemblablement passer cette durée à plusieurs jours !

les déplacements dans le champ de vision durant le cycle d'une minute. Mais nous avons déjà évoqué le refus du comité pour ce genre de solution[19]. L'ajout d'une nouvelle action réalisable par les agents telle que celle de leur permettre de se suivre ne réglerait que très partiellement le problème.

Enumérons ici certaines modifications proposées par le comité qui, bien que ne concernant pas directement ces problèmes, les modifient légèrement. Ainsi, la vitesse et le déplacement des véhicules seront calculés à la seconde par le simulateur de trafic[20] ; la vitesse maximale sera réduite de 57 à 40 ou 30 km/h[21] ; une distance de sécurité sera maintenue dans le cas où plusieurs véhicules se suivraient ; tout véhicule devra s'arrêter s'il rencontre[22] :

- un véhicule à l'arrêt, tel qu'une brigade de pompiers en train d'éteindre un incendie... ;
- le noeud précédent le bâtiment-destination ;
- un carrefour, pour lequel la prochaine route à emprunter par l'agent est plus étroite que celle qu'il utilise ;
- des débris situés au centre de la route.

7.2.2 L'agent myope

Les problèmes que nous allons mentionner ici concernent principalement les déplacements des agents. Nous verrons toutes les astuces nécessaires à mettre en oeuvre pour arriver simplement à faire se mouvoir un agent d'un endroit à un autre de la carte.

Le premier problème que nous désirons soulever concerne la manière dont le simulateur gère la vision des objets : un agent ne voit un objet que lorsque le centre de celui-ci se trouve dans son champ de vision. Sinon, il ne le voit pas du tout. Ceci pose principalement problème lorsqu'une route trop longue est impraticable. Par exemple, si elle est longue de 25 mètres, un agent ne pourra jamais, d'une de ses extrémités, l'apercevoir[23]. Il faudra qu'il se déplace dessus. Or, si elle est impraticable, l'agent ne saura justement pas s'y

[19] Voir le problème lié au champ *positionHistory*, au point 7.1.1 , page 73.

[20] Voir [Proposal, 2001], proposition 1, point I.(4). Rappelons que ceci ne change rien pour les agents au niveau des informations qu'ils vont recevoir, car le changement du pas d'incrémentation d'un module n'affecte en rien les autres modules. Le lecteur qui désirerait se rafraîchir la mémoire à ce sujet est invité à relire le point 4.2, page 32.

[21] Voir [Proposal, 2001], proposition 1, point I.(13).

[22] Voir [Proposal, 2001] proposition 1, point I.(4).

[23] Son champ de vision est de 10 mètres, alors que le centre de la route est à 12.5 mètres.

déplacer pour s'en rendre compte[24]. Ainsi, un agent n'aura jamais la possibilité de voir qu'une route de plus de 20 mètres de long est impraticable.

Le second problème provient d'un bogue du simulateur de trafic, qui ignore les déplacements demandés par les agents lorsque ceux-ci vont d'une arête (route) au noeud adjacent (carrefour). Lorsqu'un tel déplacement doit être exécuté, il est alors obligatoire de rallonger artificiellement le chemin pour que le déplacement soit pris en compte. Sinon, il est simplement ignoré.

Voici le troisième point qui peut poser problème. Lorsqu'un agent fournit au simulateur la demande de se déplacer le long d'un chemin contenant une route impraticable, le simulateur le déplace jusqu'à l'arête du graphe précédant cette route. Or pour déblayer une route, un agent policier doit obligatoirement se trouver sur le noeud adjacent à cette route. Dès lors, un agent policier qui se verrait arrêté sur une arête précédent une route obstruée et qui désirerait la déblayer devrait alors demander un déplacement de son arête au noeud adjacent... ce qui nous mène directement au problème précèdent.

Enfin, le dernier point, qui empêche de maîtriser facilement les problèmes précédents, concerne l'absence d'information communiquée par le simulateur lorsqu'il ignore le déplacement d'un agent. S'agit-il d'une demande arrivée au noyau trop tardivement ? Ou bien le chemin est-il trop court et devrait-il être rallongé ? Et si la rallonge contenait justement une route obstruée de plus de 25 mètres de long, que notre agent ne saurait détecter ? On voit que les problèmes qui peuvent se présenter ici ne sont pas toujours aisés à gérer et leur combinaison peut s'avérer un véritable casse-tête pour le développeur mal informé[25].

Le comité s'est bien rendu compte de ces problèmes et a proposé de modifier le simulateur pour permettre aux agents de voir une route bloquée lorsque celle-ci dépasse 20m de long[26]. De plus, les versions futures du noyau devraient permettre aux agents de connaître la raison pour laquelle ils ne se

[24]Rappelons que dans cette version du simulateur, les agents se déplacent soit sur les noeuds (carrefours), soit sur les arêtes (point central des routes). Rappelons aussi que la quantité de débris est une propriété d'une route. Il faut donc pouvoir «voir» la route pour savoir si elle est praticable ou non.

[25]Notons ici déjà que, comme souvent, ces problèmes ne sont renseignés nulle part. Nous reparlerons de ce problème de manque d'informations au point 7.3

[26]voir [Proposal, 2001], proposition 1, point I.(5).

sont pas déplacés comme ils le désiraient[27].

7.2.3 L'agent paranormal

Il arrive fréquemment que lors des premières minutes de la simulation, tous les civils s'échappent des bâtiments, y compris de ceux qui sont effondrés, comme s'ils pouvaient traverser les murs et les débris. Ceci n'est bien entendu pas du tout réaliste mais provient vraisemblablement d'un bogue quelconque et non d'une mauvaise approche de simulation de la réalité. Les agents paramédicaux n'ont alors plus qu'à conduire les blessés qui déambulent dans les rues aux refuges, l'état de santé de ces blessés restant relativement stable puisqu'ils n'ont plus à souffrir des décombres sous lesquels ils devraient être enterrés.

Ceci n'a bien sûr échappé à aucun membre du comité ni à aucun participant lors du championnat de 2001, lorsque certaines équipes se sont vues avantagées par ce problème[28]. Ce problème sera vraisemblablement corrigé pour le prochain championnat[29].

7.2.4 L'ordre dans la destinée

Les simulations ne sont pas déterministes, même si on fixe les conditions initiales. Ceci n'est pas un problème en soi, mais l'importance de ce non déterminisme rend la comparaison entre équipes d'agents assez difficile, car deux équipes peuvent se voir, lors de plusieurs répétitions d'une même simulation, tantôt vainqueurs et tantôt perdantes[30].

Une des raisons de ces importantes fluctuations provient du module de simulation de l'évolution de l'état de santé des agents (*Miscellaneous Simulator*). Celui-ci détermine la destinée des victimes (sur base de nombres aléatoirement générés) à partir de l'état des autres agents. Ainsi le sauvetage d'une victime affecte la destinée des autres victimes, favorisant plus ou

[27] voir [Proposal, 2001], proposition 1, point I.(10).

[28] Le lecteur peut se rendre compte de l'importance du problème en consultant les résultats de l'annexe E. Par exemple, le nombre de victimes obtenues par l'équipe YabAI, pour la première situation initiale (Comitee_map), sont compris entre 21 et 36. Une seule simulation ne donne que 12 victimes, et fut obtenue grâce à ce problème.

[29] Voir [Proposal, 2001], proposition 1, point VI.(4).

[30] Le lecteur peut s'en rendre compte en consultant l'annexe E. Par exemple, lors de la première situation initiale (comitee_map), notre équipe voit ses résultats varier de 17 à 28, contre 21 à 37 pour YabAI (nous n'avons pas inclus le 12 causé par le problème du point 7.2.3).

moins une équipe en fonction de l'ordre dans lequel elle sauve les agents. Le comité propose de résoudre ce problème en rendant les calculs (de nombres aléatoires) indépendants de l'état de santé des autres civils[31].

7.2.5 Autres bogues

Nous présentons ici les derniers problèmes de cette section. Il s'agit premièrement de deux (groupes de) bogues, qui rendent la simulation complètement inutilisable lorsqu'ils se présentent. Il arrive en effet, pour des raisons que nous ne connaissons pas, et à n'importe quel moment, que le simulateur de trafic ou le noyau se «plantent», c'est-à-dire qu'ils s'arrêtent de fonctionner purement et simplement. Nous imaginons que ces problèmes sont connus du comité, bien qu'ils ne soient pas présents dans le document [Proposal, 2001].

Lorsque le problème provient du simulateur de trafic, les déplacements sont ignorés, et donc plus aucun agent ne peut bouger. Cependant, tout le reste de la simulation reste opérationnel : le temps avance, les incendies continuent de s'étendre (les pompiers peuvent même les éteindre s'ils sont à proximité), l'état de santé des agents continue de s'aggraver, etc. Ce problème est une des raisons qui empêchent l'automatisation des simulations pour la récolte de résultats, car il impose la surveillance de la simulation et l'arrêt de celle-ci sous peine d'obtenir des résultats faussement catastrophiques.

Lorsque c'est le noyau qui s'arrête de fonctionner, c'est le temps qui se fige et, par conséquent, toute la simulation. Dans ce cas, il est impossible que la simulation se termine d'elle-même. Elle ne peut donc pas donner de mauvais résultats mais il faudra cependant la redémarrer.

Le dernier problème qui nécessite une attention particulière lorsqu'on effectue une simulation concerne la fenêtre de visualisation fournie avec le simulateur. Nous nous sommes rendus compte qu'il arrivait que cette fenêtre ne prenne pas en compte certaines informations. En effet, nous avons vu un civil survivre durant toute une simulation dans un bâtiment en feu. Après avoir analysé les traces de la simulation, nous avons pourtant trouvé le moment où cet agent civil décéda.

En réalité, le noyau n'envoie jamais, après l'initialisation, le contenu du monde. Il n'envoie que ce qui y a changé. Or, comme un agent ne meurt qu'une fois, cette information n'est envoyée qu'une fois, lorsque cela arrive.

[31] Voir [Proposal, 2001], proposition 1, point VI.

Et si la fenêtre de visualisation n'en prend pas note à ce moment là, elle n'aura plus jamais l'occasion de savoir qu'il est mort et le considérera vivant jusqu'à la fin de la simulation.

Une conséquence directe de ce problème concerne l'évaluation de la simulation. Effectivement, le noyau n'envoie pas sa valeur, elle est en fait calculée par la fenêtre de visualisation. Or, comme il arrive que celle fournie avec le simulateur fasse quelques erreurs, la valeur de la fonction d'évaluation donnée par la fenêtre risque fort de ne pas être correcte. Le moyen que nous avons utilisé pour nos résultats fut d'observer les simulations à partir de leurs traces, et non plus à partir des informations envoyées directement par le noyau. En effet, la fenêtre de visualisation ne semble pas faire d'erreur lorsqu'elle lit les traces de simulation.

Nous avons communiqué ce problème au comité, cependant le document [Proposal, 2001] n'en fait pas mention. Néanmoins, lors d'une communication électronique avec M. Takeshi, membre du comité pour le championnat de 2002[32], ce dernier prit le problème au sérieux. En effet, cette fenêtre de visualisation fut également utilisée lors du championnat de 2001. Il est dès lors fort probable que ce problème soit résolu pour le prochain championnat.

7.3 Problèmes de documentation

Nous désirions terminer ce chapitre par ce dernier point, qui, selon nous, est le problème le plus important de la RoboCupRescue. Il s'agit d'une insuffisance générale d'informations. Il existe heureusement une *mailing list* qui permet aux chercheurs d'obtenir les informations qu'ils désirent, bien qu'elle ne soit à notre avis pas suffisante. Nous avons personnellement bénéficié de l'aide précieuse que nous a apporté M. Takeshi, sans qui nous n'aurions très probablement pas pu maîtriser aussi efficacement le sujet.

Le premier point qui nous a surpris, bien qu'il ne nous ait pas dérangé puisque nous avons récupéré des agents déjà fonctionnels[33], fut les erreurs contenues dans le manuel d'utilisation du simulateur [Committee, 2000]. Celui-ci comporte notamment des erreurs concernant des constantes de base nécessaires à la communication de bas niveau avec le noyau. De plus, il n'existe encore à ce jour aucun *erratum*. Il n'existe pas non plus, à notre connaissance,

[32] Rappelons que M. Takeshi est le réalisateur de YabAI.

[33] Ce qui ne fut pas le cas de certains de nos camarades traitant du même sujet et désireux de commencer leurs agents à partir de zéro.

de mise en garde signalant que des modifications ont été apportées aux informations présentes dans ce manuel. Nous ne nous expliquons toujours pas cette absence de mise à jour ou de nouvelle version, durant plus de 18 mois, de ces informations fondamentales aux nouveaux chercheurs qui désireraient participer au développement de cette discipline.

Ensuite, nous ne sommes pas parvenus à trouver d'informations détaillées concernant certaines valeurs de propriétés d'objets[34]. Par exemple, les valeurs du champ *fieryness* ne sont expliquées nulle part. Nous les avons découvertes au gré de notre programmation, ainsi que dans la lecture de documents publiés où on les mentionnait brièvement en parlant de «*early fire, very early fire*» et autres. Voici toute l'information disponible dans le document de référence [Committee, 2000] : «*fieryness : integer* [0~7FFFFFFF] ; *how much it burns.*».

D'autres informations qui auraient pu être répertoriées plus tôt, sont celles concernant les problèmes techniques que nous avons présentés au point 7.2. En effet, lors de la réalisation du stage, aucun document ne permettait d'avoir connaissance de ces problèmes[35], avant l'apparition du document [Proposal, 2001]. Ces problèmes étant antérieurs au championnat de 2001, ce manque d'information a couvert au minimum une durée de 6 mois, voire de près d'un an. Durant cette période, de nombreux chercheurs auront vraisemblablement perdu du temps à identifier et comprendre des problèmes déjà connus.

[34] C'est le cas pour les propriétés *buriedness, repairCost, damage, buildingCode, etc.*

[35] Nous en avons découvert la majorité en lisant le code de M. Takeshi, qui les a lui-même découverts en analysant le code des différents modules du simulateur. Ce dernier aurait par ailleurs averti le comité de ces problèmes avant le championnat de 2001 (il ne faisait alors pas encore partie du comité). Le comité n'en aura malheureusement fait mention nulle part.

Chapitre 8

Simuler des tempêtes de verglas

8.1 Introduction

Nous avons souvent souligné le caractère modulable du simulateur de la RoboCupRescue, en laissant peut-être sous-entendre qu'il suffirait de changer l'un ou l'autre module pour pouvoir l'adapter à tout type de situation. Nous nous sommes alors interrogés sur la faisabilité d'un tel changement et sur la difficulté que cela représenterait.

Nous sommes bien conscient qu'un réel travail d'adaptation du simulateur à un autre type de désastre serait un travail de grande ampleur, ne fut-ce que l'analyse théorique qui lui serait nécessaire. Un tel travail pourrait vraisemblablement faire l'objet d'une thèse de doctorat. Cependant, plutôt que de terminer notre ouvrage en bouclant simplement notre état de l'art, nous désirons créer une ouverture vers d'autres possibilités de développements et d'évolutions dans un avenir proche. C'est pourquoi nous proposons au lecteur une ébauche du travail théorique qui pourrait être réalisé au sujet d'une catastrophe naturelle un peu particulière dont a souffert récemment le Québec, à savoir une tempête de verglas[1].

Nous avons donc organisé ce chapitre en une étude préliminaire des tempêtes de verglas, où nous expliquerons comment ce phénomène se produit, ainsi que les sinistres qu'il peut causer. Ensuite, nous présenterons les adaptations du modèle du simulateur de la RoboCupRescue pour prendre en compte les particularités de ce genre de tempêtes. Nous terminerons en traçant les

[1] Notre travail se base sur le rapport scientifique [Les Publications du Québec, 1999] publié par le gouvernement du Québec, ainsi que sur une interview, menée par nos soins, d'un responsable de la sécurité civile du Québec.

grandes lignes de nouveaux modules qui pourraient vraisemblablement être nécessaires.

8.2 Etude du sinistre

8.2.1 Tempêtes de verglas en général

Une tempête de verglas n'est pas, comme pourrait le croire le lecteur, une tempête de glace. En effet, il ne s'agit pas de morceaux de glace ou de grêle tombant du ciel et créant des ravages au moment de leur impact, mais d'une pluie un peu particulière, appelée «pluie verglaçante», qui gèle instantanément (et se transforme en verglas) au moment de son contact avec le sol ou avec tout autre objet sur lequel elle vient se déposer.

FIG. 8.1 – Photo amateur après une pluie verglaçante

Ce phénomène arrive lorsqu'un front chaud chargé d'humidité rencontre un front froid et sec, et que la température au sol est inférieure à zéro degré Celsius. Le front froid glisse en dessous du front chaud, faisant précipiter l'humidité du front chaud situé au dessus de lui. Les gouttes d'eau, en traversant le front froid, se refroidissent alors progressivement jusqu'à atteindre une température inférieure à zéro degré. Elle sont à ce stade dans un état de

«surfusion[2]». Ainsi, dès que ces gouttes d'eau atterrissent sur un objet, elles se solidifient immédiatement pour former de la glace. Dans les jours qui suivront, le front froid remplacera le front chaud, faisant tomber la température très rapidement. Habituellement, elle atteindra 20 à 25 degrés en dessous de zéro, et il n'est pas rare qu'elle dépasse les -30.

Une tempête de verglas est donc un phénomène relativement graduel qui contraste avec d'autres désastres beaucoup plus brutaux. Dans la mesure où le verglas ne cause généralement pas de dégâts majeurs aux bâtiments et aux infrastructures, il n'est pas immédiatement perçu comme une menace aux biens comme peuvent l'être un tremblement de terre, un ouragan ou même une inondation. Par ailleurs, le phénomène de pluie verglaçante n'est pas exceptionnel au Québec. Par exemple, dans la région de Montréal, on l'observe en moyenne une douzaine de fois par hiver. Cependant, ce phénomène se maintient rarement sur une période prolongée car les masses d'air ne restent habituellement pas stationnaires.

8.2.2 Le verglas de 1998

Les 5, 7 et 8 janvier 1998, trois chutes de pluie verglaçante déposèrent, sur toute surface exposée dans une superficie de 40 000 km^2 au sud du Québec[3], une couche de glace d'une hauteur allant de 5 à plus de 10 cm de hauteur. Dès le 6 janvier, ce sinistre climatique est devenu un sinistre technologique, avec la chute des premières lignes de transport et de distribution d'électricité et l'interruption des approvisionnements en énergie qui en a résulté. En conséquence, la plupart des infrastructures essentielles de la société ont connu des dysfonctionnements majeurs et la moitié de la population s'est retrouvée sans électricité[4]. L'ampleur des dégâts matériels fut sans précédent au Canada, dépassant le milliard de dollars canadiens. Le nombre de pertes de vies

[2]La «surfusion» est un état instable de l'eau liquide à une température inférieure à 0 degré Celsius.

[3]Principalement dans la région de Montréal, mais aussi dans celles de l'Estrie, de l'Outaouais, de Chaudière-Appalaches, de Laval, de Lanaudière, des Laurentides, de la Montérégie et du Centre-du-Québec. Plus précisément, ce territoire comprend 598 des 1 421 municipalités du Québec. La population totale de ces municipalité est de 4 826 586 habitants, soit 66,9% de la population Québécoise.

[4]L'évolution du nombre de foyers s'étant retrouvés sans électricité passa de 700 000 le 6 janvier, à 1 400 000 le 9 janvier, ce qui représente environ la moitié de la population du Québec, à savoir 3,5 millions de personnes. Il faudra attendre le 5 février pour que la situation soit complètement rétablie.

quant à lui semble difficile à évaluer[5].

Voici chronologiquement comment le sinistre s'installe. Au début, les services de la ville commencent par épandre du sel, tentant de maintenir le réseau routier effectif. Une à une, les autoroutes fermeront, menacées par la chute d'importantes stalactites de glace situées sous les ponts et sous les panneaux surplombant l'autoroute, et qu'il faudra briser à l'aide d'échelles et de maillets. Un peu plus tard, des branches d'arbres casseront sous le poids de la glace et tomberont alors dans la neige recouvrant le sol. Ils abîmeront et casseront progressivement les souffleuses de neige[6], diminuant graduellement la capacité de déblayage du réseau routier.

Des branches d'arbres s'effondreront aussi sur certaines lignes électriques, privant certains quartiers d'électricité. Des systèmes de suspension d'autres lignes se rompront, provoquant la chute au sol de ces câbles et créant des courts-circuits. Plus tard encore, ce seront des pylônes électriques qui ploieront ou s'effondreront, soit sous le poids de la glace accumulée sur leur structure, soit sous la tension exercée par leurs câbles[7]. La pluie continuera de tomber, jusqu'au moment où ils s'effondreront en cascade, comme un véritable jeu de domino[8]. En 1998 en particulier, pratiquement toute la structure des pylônes de la ceinture autour de Montréal (900 pylônes) s'est effondrée.

[5]Le Bureau du coroner estime que seuls trente décès sont attribuables aux évènements reliés au verglas (Chute de toit, brûlures, intoxications au CO et hypothermie). On remarque cependant que malgré les conditions qui prévalaient sur les routes, aucun accident mortel n'a été attribué au verglas. D'un autre côté, on observe, uniquement dans les régions touchées, une augmentation du nombre total de décès en janvier 1998 par rapport à janvier 1997. La Montérégie a, par exemple, connu près de deux cent décès de plus que le même mois de l'année précédente, soit 924 au lieu de 733.

[6]Chaque année, il tombe au Québec plus de deux mètres de neige. De simples chasse-neige ne sont plus suffisants pour déblayer le réseau routier, et il faut utiliser de grosses machines, appelées «souffleuses de neige», qui aspirent la neige sur les routes et la souffle dans des camions-benne, qui vont alors la déverser dans les fleuves de la région.

[7]La glace se dépose sur la partie supérieure des fils et entraîne une rotation lente de ceux-ci, se déposant alors tout autour, tel un manchon. Au total, la glace ainsi accumulée sur les câbles devient beaucoup plus importante que celle accumulée sur les objets fixes, rendant la tension exercé par les câbles de plus en plus forte sur les poteaux auxquels ils sont attachés. De plus, il arrive que les attaches de ces câbles se rompent, créant une rupture d'équilibre des charges.

[8]La propagation de l'écroulement des pylônes ou des portiques en bois est généralement stoppée lorsqu'elle rencontre des structures plus robustes tel qu'un pylône d'angle. Il s'agit ainsi de parties du réseau qui s'écroulent ensembles et non tout le réseau d'un coup.

FIG. 8.2 – Ces photos permettent d'apprécier le volume de glace qui se dépose sur les pylônes, provoquant leur effondrement.

8.2.3 Les conséquences du verglas

Les impacts directs du verglas sont multipliés par les conséquences de l'interruption des approvisionnements en électricité sur de nombreuses infrastructures. Dans des milliers de résidences, les pannes d'électricité entraînent l'arrêt du chauffage, l'impossibilité de s'éclairer ou de faire cuire sa nourriture. Les aliments périssent ou gèlent. Les canalisations gèlent ou crèvent, ce qui dans ce dernier cas crée des inondations.

La circulation urbaine est rendue extrêmement difficile, car en plus du verglas sur les routes, les fils électriques et les branches (voir les arbres eux-mêmes) jonchent un grand nombre d'artères. Les feux de circulation sont inopérants, les voies non éclairées et le pompage de l'eau dans les tunnels et sous certains viaducs arrêtés. Certaines routes et autoroutes sont fermées[9]. La production et la distribution de produits pétroliers présentent des perturbations majeures[10]. Une pénurie financière s'installe en raison de l'arrêt des nombreux guichets automatiques et succursales bancaires. Les infrastructures endommagées des compagnies de télécommunications retardent ou empêchent les communications, donnant encore un sentiment accru d'isolement pour les sinistrés. L'approvisionnement en eau connaît des difficultés en raison des pannes des usines de filtration[11]. Enfin, les centres hospitaliers connaissent eux aussi de graves dysfonctionnements.

Lorsque les pylônes s'écrasent, la tension générale baisse jusqu'au moment où la demande dépasse l'offre. A cet instant la surcharge fait sauter toute la distribution d'électricité de la ville. Il faut demander aux civils, qui veulent tester si l'électricité est revenue, d'arrêter de jouer avec leurs interrupteurs, car cela crée une demande massive, qui recrée une surcharge à chaque fois qu'on tente de rebrancher l'électricité. On leur demande aussi d'utiliser le minimum d'électricité nécessaire à leur survie pour arriver à maintenir un minimum d'électricité disponible pour les centres qui en ont le plus besoin (station de pompage et de gestion de l'eau, hôpitaux, etc.). On essaye aussi de garder les entreprises en marche. En effet, si une entreprise n'a plus d'élec-

[9] Pour dégager une voie publique, il faut enlever les débris, débrancher les câbles tombés mais encore «vivants», et puis faire passer des souffleuses de neige, des chasse-neiges et des épandeurs de sels avant de pouvoir réouvrir la route. Il a fallu parfois jusqu'à une semaine pour dégager une route.

[10] En 1998, seul un dépôt utilisant une rame de chargement fonctionnant avec génératrice put continuer à distribuer des produits pétroliers à Montréal. Cependant, ce dépôt ne disposait que d'une réserve de deux jours pour répondre aux besoins de la population.

[11] Beaucoup d'efforts ont été déployés lors de la tempête pour garder la station de pompage d'eau en marche. Sans eau, Montréal aurait dû être évacuée.

tricité, elle risque de perdre beaucoup d'argent, voire de tomber en faillite et de provoquer des milliers de pertes d'emploi. Pour cette même raison, le *Just-In-Time* nécessite de garder un maximum de routes ouvertes.

Il faut héberger les gens sans électricité[12] et donc aménager des écoles[13] et d'autres bâtiments en centres d'hébergement. Malheureusement, certaines personnes refusent de quitter leur domicile. Il faut donc leur amener du bois de chauffage pour éviter qu'ils périssent de froid. Il est aussi demandé aux civils de ne pas quitter leur domicile de leur propre chef, mais il y en a toujours qui ne respectent pas ce conseil et qui vont se mettre dans des situations qui demandent d'aller les secourir. Des problèmes de pillages ont aussi lieu. Les policiers, aidés des pompiers, doivent sécuriser le quartier, établir des couvre-feux...

8.3 Adaptation du modèle

8.3.1 Une approche globale

Nous proposons tout d'abord de résumer les différentes conséquences qui ont été retenues dans le modèle du simulateur de la RoboCupRescue. Un tremblement de terre se déclare et fait s'effondrer plusieurs bâtiments. De ces bâtiments tombent un certain nombre de débris sur les chaussées. Certains incendies se déclarent dans divers endroits de la ville. Des civils sont prisonniers sous des décombres ou gravement brûlés par les incendies et leur état de santé s'en trouve influencé. Des refuges sont aménagés pour y conduire les civils.

Si nous analysons sommairement les conséquences des tempêtes de verglas, nous pouvons en retirer ce qui suit. Des chutes de pluie verglaçante déposent de la glace sur toute structure exposée et sur la chaussée. Suite à cela, des branches d'arbres ou les arbres eux-mêmes ainsi que des câbles électriques tombent sur la chaussée. Les pylônes s'effondrent sous leur propre poids. Une partie de la population se retrouve isolée du réseau de distribution d'électricité, sans source d'énergie pour se chauffer, pour cuire leurs aliments, etc. Leur état de santé se détériorera au fil du temps (hypothermie). Des refuges seront aménagés pour les accueillir.

[12] On considère qu'un civil peut tenir 48 heures sans chauffage lorsque la température descend en dessous de 20 degrés à l'extérieur.

[13] Environ 300 écoles furent transformées en refuges lors du sinistre de 1998.

8.3.2 Modifications nécessaires

Dans l'ensemble des propriétés qui ont été attribuées aux objets de la simulation[14], certaines n'ont plus réellement de raison d'être. Ainsi, la propriété d'ensevelissement des civils ainsi que celle d'effondrement des bâtiments ne sont plus nécessaires, puisque les tempêtes de verglas n'affectent pratiquement pas les bâtiments. De même, elles ne provoquent pas d'incendie, c'est pourquoi toutes les propriétés liées aux incendies (*ignition, fieryness, waterQuantity* et *stretchedLength*) ne sont plus utiles.

Inversement, d'autres propriétés doivent être rajoutées, pour permettre de représenter l'accumulation de glace, ainsi que le réseau électrique. Pour les bâtiments, une propriété renseignant la présence d'électricité sera nécessaire ainsi que celle de leur consommation en électricité. Une dernière propriété pourra être ajoutée aux routes concernant la couche de glace qui s'y sera déposée.

Il sera alors nécessaire de créer de nouveaux objets, à savoir les lignes électriques et les pylônes. Ceci pourrait se faire en considérant un pylône comme un noeud particulier du graphe (comme le sont les bâtiments) et un câble comme une arête (à la manière des routes). Voici, sous forme de tableaux, nos propositions pour modéliser ces objets :

Pylône

Propriété	Domaine	Commentaire
x	entier 32 bits non négatif	Coordonnée x du noeud, en millimètre, à partir de l'origine.
y	entier 32 bits non négatif	Coordonnée y du noeud, en millimètre, à partir de l'origine.
hauteur	entier 32 bits non négatif	Hauteur du pylône.
type	entier 32 bits non négatif	Représente le type de pylône. Par exemple, 0 pourrait représenter un portique en bois, et 1 un pylône en acier.
effondrement	entier : $0 \sim 100$	Représente le degré d'effondrement. Les valeurs vont de 0 (non effondré) à 100% d'effondrement.

[14]Nous parlons ici des propriétés du SIG, présentées à l'annexe A.

connexions	liste d'IDs	Liste des IDs des objets normalement connectés au pylône.
attaches	liste d'entiers 32 bits non négatif	Liste de nombres représentant l'état des attaches. 0 représente une attache rompue et 1 une intacte. L'ordre de ces nombres est identique à celui de la propriété connexions.
glace	entier 32 bits non négatif	Hauteur, en millimètre, de la couche de glace déposée sur la structure.

Nous pouvons considérer que le type des attaches est représenté implicitement par le type de pylône. Nous pourrions encore, à la manière du modèle des bâtiments, ajouter à cette liste des informations concernant la forme du pylône, sa hauteur, sa surface, etc.

Ligne électrique (entre 2 pylônes)

Propriété	Domaine	Commentaire
tête	ID du pylône	ID du point normalement extrémité initiale de la ligne électrique.
queue	ID du pylône	ID du point normalement extrémité finale de la ligne électrique.
contacts	liste d'IDs	liste des IDs des objets que touche le câble, excepté ses extrémités. Cette liste devrait être vide tant que le câble ne s'est pas détaché ou rompu.
état	entier 32 bits non négatif	Représente l'état du câble. Par exemple, 0 signifie qu'il est en bon état, 1 et 2 qu'il s'est détaché à la tête ou à la queue, et 3 qu'il s'est rompu.
longueur	entier 32 bits non négatif	Longueur de la ligne électrique. L'unité de mesure est le millimètre.
type	entier 32 bits non négatif	Représente le type de ligne électrique : 735 kV, 315kV, 230kV ou 120kV.
électricité	entier : 0 ou 1	Représente la présence d'électricité (1) ou non (0).

glace	entier 32 bits non négatif	Epaisseur, en millimètre, de la couche de glace déposée autour la ligne.

On peut encore imaginer ajouter une propriété semblable à *repairCost*, pour connaître le temps nécessaire au déglaçage et au rétablissement de la ligne. D'autres propriétés pourraient aussi être ajoutées pour renseigner plus exactement l'endroit auquel le câble s'est rompu, et si c'est le cas, quelle extrémité de la rupture touche quel objet (amélioration de la propriété contacts).

8.3.3 La gestion du temps

Alors qu'un tremblement de terre ne dure pas plus de quelques minutes, une tempête de verglas s'étale sur plusieurs jours. C'est pourquoi nous pensons que simuler 5 heures après le désastre n'est pas une durée suffisante. Cependant, le phénomène d'une tempête de glace étant plus lent, le pas d'incrémentation du temps pourrait être revu à la hausse. En effet, alors qu'un incendie peut évoluer rapidement en quelques minutes en fonction du vent et de son voisinage, l'état futur du réseau électrique n'est influencé que par l'accumulation lente de glace sur ses structures de transport. Il en va de même pour l'état des civils : alors qu'un civil ne peut généralement tenir que quelques heures lorsqu'un bâtiment s'est effondré sur lui, un civil sans électricité pourra tenir 48 heures malgré une température extérieure avoisinant les -30 degrés Celsius.

D'autre part, simuler le début d'une tempête de verglas ne présente pas énormément d'intérêt, puisque les problèmes importants ne surgissent que lorsque le réseau électrique est touché. Dans le cas de la tempête de 1998, celui-ci fut affecté le 6 janvier, soit après la première pluie verglaçante. C'est pourquoi la simulation devrait commencer au début de la deuxième vague de pluie verglaçante, avec 20 à 40 millimètres de glace déjà répartie sur les différentes structures. Nous pouvons alors simuler une durée d'environ 3 jours. Si nous fixons cette durée à 75 heures et le pas d'incrémentation du temps à un quart d'heure par cycle, nous arrivons au même nombre de cycles, ce qui présente l'avantage de garder une durée (en temps réel) de simulation équivalente.

Cependant, un problème qui risque de se poser plus en avant concerne les mouvements des agents. Nous avions déjà critiqué le pas d'incrémentation du temps lors des simulations de tremblement de terre, trouvant celui d'une minute trop élevé. Même si les véhicules se déplaçaient moins rapidement,

un pas d'incrémentation 15 fois plus élevé diminuerait fortement la quantité d'information qu'ils recevront sur leur monde et rendrait cette problématique plus importante encore. Notre suggestion reste cependant la même : le noyau devra envoyer les informations concernant les évènements remarquables qui se seront passés lors de cet intervalle de temps.

8.3.4 Les équipes de secours

En ce qui concerne l'adaptation des agents présents dans la RoboCupRescue, nous pouvons garder le même nombre de type d'agents et même en récupérer certains. Pour commencer, chacun des trois types d'équipes de secours sur le terrain pourra répondre d'un quartier général, comme c'est déjà le cas dans le simulateur.

Ensuite, comme dans le cas des tremblements de terre, la situation requiert des équipes qui se chargent de dégager le réseau routier de ses décombres. Notons ici que le phénomène des pluies verglaçantes n'est pas un phénomène ponctuel comme c'est le cas d'un tremblement de terre. La glace devra pouvoir s'accumuler sur les structures tout au long de la simulation, causant des dégâts à tout moment, entre autre en précipitant des branches d'arbres sur la voirie. L'épandage sera bien entendu inclu dans leur travail, mais pourra être considéré comme partie intégrante du déblayage global des routes.

Les équipes de sauvetage de civils sont toujours nécessaires. Celles-ci n'auront plus le devoir de déterrer des civils, mais devront toujours venir les chercher dans leur bâtiment afin de les conduire dans des refuges.

Pour finir, les pompiers ne seront plus nécessaires. Cependant, des équipes d'électriciens devraient les remplacer, afin d'empêcher l'état du réseau électrique de s'aggraver (en cassant la glace accumulée sur les fils par exemple). Ils auront aussi comme mission de réparer les dégâts du réseau et de rétablir l'électricité.

8.4 Les différents modules

Maintenant que nous avons établi le cadre général, nous pouvons enfin analyser le rôle des différents modules, et plus particulièrement des différents simulateurs de composants. Pour rappel, ils sont au nombre de 5 dans le simulateur de la RoboCupRescue :

- un simulateur de tremblements de terre ;
- un simulateur d'incendie ;
- un simulateur d'accumulation de débris sur les routes ;
- un simulateur de trafic routier ;
- un simulateur d'évolution de santé.

Le simulateur de tremblement de terre ne nous est évidemment plus utile. De même, nous avons déjà expliqué que le simulateur d'incendie n'est plus nécessaire.

L'accumulateur de débris sur les routes devra être adapté pour tenir compte du fait qu'il ne s'agit plus de débris de bâtiments qui jonchent les routes, mais de glace et de branches d'arbres. Plus généralement, nous devrions avoir un module qui calcule, en fonction de la couche de glace présente et de leur résistance, le moment de rupture des câbles électriques ou de leurs attaches, ainsi que celui de l'effondrement des pylônes et des arbres .

Nous pouvons garder le simulateur de trafic routier tel quel, en veillant toutefois à adapter la vitesse des véhicules en fonction de la couche de glace sur la route. Il en va de même pour le simulateur d'évolution de santé, qui devra être adapté aux cas d'hypothermie.

Terminons alors notre ébauche analytique des tempêtes de glace par le nouveau simulateur de composant qui devra être crée : le simulateur de réseau électrique. Ce simulateur sera responsable du transport de l'électricité dans la ville. En effet, il devra propager les pannes, prendre en compte les court-circuits, utiliser des voies ou circuits alternatifs pour continuer à approvisionner les bâtiments en électricité. Il devra également être capable de calculer la tension sur les circuits en fonction de la consommation des bâtiments et de créer les surcharges lorsque cette dernière sera trop importante. Il pourra aussi être responsable du calcul de la consommation des bâtiments en fonction du type de bâtiment et du nombre de civils y étant présents.

8.4.1 Un avenir...

Nous avons présenté comment, à partir de conditions météorologiques particulières, de simples gouttes de pluie pouvaient provoquer l'effondrement des réseaux électrique et routier d'une ville avec toutes les conséquences que cela peut impliquer. Sur base de ces constatations, nous avons proposé une adaptation du modèle de représentation interne d'une ville du simulateur de la RoboCupRescue. Nous avons par ailleurs discuté de l'évolution sensiblement plus lente du sinistre au cours du temps, ainsi que du rôle qu'auraient

à jouer les agents sauveteurs dans un tel sinistre. Nous avons alors terminé notre analyse par certaines suggestions concernant les adaptations des simulateurs de composant existants ainsi que la création d'un module de simulation de réseau électrique.

Nous espérons, à travers ce chapitre, avoir montré la faisabilité d'une adaptation du simulateur à un genre tout à fait différent de catastrophe naturelle. Partant de là, nous pouvons imaginer d'autres adaptations faisant intervenir d'autres phénomènes catastrophiques, naturels ou même créés par l'homme.

En supposant que la RoboCupRescue parvienne un jour à remporter son défi et à voir son simulateur, relié à des systèmes d'acquisition de données sur le terrain, servir comme système fiable d'aide à la décision, ce sera alors probablement une multitude de désastres qu'elle nous permettra de gérer efficacement...

Chapitre 9

Conclusion

Ici s'achève notre immersion dans les mondes simulés et leurs habitants, les agents. Après avoir présenté ce que sont les agents et les systèmes qu'ils forment, nous nous sommes penchés sur leur classification. Nous avons alors fait la présentation d'une organisation qui s'y intéresse tout particulièrement, à savoir la RoboCup. Après avoir présenté ses différents pôles d'activités, nous nous sommes arrêtés sur un de ses projets dont le but est de recréer par simulation le phénomène des tremblements de terre, sur base du tragique sinistre qui a secoué la ville de Kobe en 1995. Ce travail de simulation nous a particulièrement intéressé, aussi l'avons-nous exploré en profondeur pour en comprendre les rouages, organisés en modules relativement indépendants. Nous en avons alors examiné la population d'agents, à travers ses pouvoirs d'actions et d'organisations, avant de nous pencher sur une équipe d'agents sauveteurs toute particulière, nommée YabAI. Reconnue en 2001 comme meilleure en qualité de gestion de désastre simulé par la RoboCupRescue, nous l'avons utilisée comme base de travail pour la création de nos propres agents. Nous en avons analysé et présenté les règles de décisions, avant d'édifier les nôtres. Nous avons alors réalisé une étude comparative entre ces deux stratégies, qui sembla conclure par de meilleurs résultats en notre faveur. Cependant, estimant qu'un bon nombre d'améliorations devrait être apporté au simulateur avant de tirer des conclusions hâtives, nous nous sommes autorisés quelques critiques qui pourraient permettre d'améliorer le réalisme du simulateur. Enfin, désireux d'ouvrir le champ des possibilités que nous offre ce simulateur, nous avons souhaité apporter un survol des adaptations qui seraient nécessaires afin de l'appliquer à un genre tout particulier de catastrophes naturelles, à savoir les tempêtes de verglas.

A travers ce parcours, nous espérons avoir suscité de l'intérêt auprès du lecteur pour les systèmes multiagents, comme ce fut le cas pour nous tout au

long de cette année.

D'ici une cinquantaine d'années, la RoboCup espère avoir créé une équipe de robots capables de gagner une rencontre amicale de football contre la meilleure équipe du monde. Ces robots sont actuellement pilotés par des agents, et le seront probablement toujours alors. Imaginez comment pourrait évoluer le monde d'ici là. On trouve de plus en plus d'agents sur Internet, et de plus en plus Internet dans nos vies, que ce soit dans nos téléphones cellulaires ou dans nos télévisions. Pourrait-on imaginer que les avancées de l'intelligence artificielle nous permettent de côtoyer au quotidien cette nouvelle population virtuelle qu'elle crée, et qui nous semble présenter de plus en plus de signes d'intelligence ?

Comme l'écrivait Henri Bergson en 1940 dans son livre intitulé *L'évolution créatrice*, «Notre intelligence, telle qu'elle sort des mains de la nature, a pour objet principal le solide inorganisé». Au vu des premiers résultats que donne la coordination d'agents artificiellement intelligents, nous pourrions terminer en ajoutant que l'intelligence, telle qu'elle sort des mains de l'Homme, a pour objectif final l'abstrait organisé...

Annexe A

Informations manipulées par le noyau

Cette annexe décrit les objets du monde simulé. Elle est issue du manuel du simulateur RoboCupRescue [Committee, 2000]. Les objets principaux sont les familles de civils, les bâtiments et les routes. La majorité des autres objets en sont dérivés. Les objets inanimés sont représentés par un graphe, dont les noeuds sont des bâtiments, des carrefours et autres points de connexion, tandis que les arêtes sont des routes et des rivières. Un identifiant unique (un entier positif) est assigné à chaque objet du monde.

A.1 Le monde

La longitude et la latitude réelles sont utilisées pour localiser la position des objets dans le Système d'Information Géographique (SIG). La donnée concernant les positions des objets simulés est traduite dans un nouveau système de coordonnées dont l'origine est un point de la région simulée.

Propriété	Domaine	Commentaire
startTime	entier 32 bits[1]	L'heure du début de la simulation (en minutes depuis le 1er janvier 1970 à 0h00).
windForce	entier 32 bits non négatif	La force actuelle du vent. L'unité est le millimètre/heure.

[1] Pour information, la valeur d'un entier 32 bits s'étend, en valeur absolue, de 0 à $2^{31}-1$, soit 2 147 483 647

Propriété	Domaine	Commentaire
windDirection	entier 32 bits non négatif	Direction actuelle du vent. La direction Nord est 0, et la valeur s'étend jusque $360 * 60 * 60 - 1 =$ 1 295 999 secondes, dans le sens des aiguilles d'une montre.
longitude	entier 32 bits : $-600\,000 \sim 600\,000$	Longitude de la nouvelle coordonnée de l'origine. Une longitude Est est positive, une longitude Ouest est négative. L'unité de mesure est la seconde.
latitude	entier 32 bits : $-600\,000 \sim 600\,000$	Latitude de la nouvelle coordonnée de l'origine. Une latitude Nord est positive, une latitude Sud est négative. L'unité de mesure est la seconde. Ces deux dernières propriétés impliquent que l'origine doit se trouver au Sud et à l'Ouest de tout point utilisé dans la simulation.

A.2 Civil (agent)

Propriété	Domaine	Commentaire
position	ID[2] de l'objet ou 0	ID de l'objet sur/dans lequel est le civil. Quand le civil est sur une route, c'est l'ID de la route. S'il est dans un bâtiment, c'est l'ID du bâtiment. Quand il est dans une ambulance, c'est l'ID de l'ambulance, etc. ID=0 signifie qu'il n'est nulle part.
positionExtra	entier 32 bits	Quand un civil est sur une route, cette propriété représente la distance qui le sépare du début de la route. Cette valeur varie entre 0 et la longueur de la route. Quand le civil est sur/dans un autre objet, cette valeur vaut 0.

[2]ID = identifiant unique attaché à un objet

Propriété	Domaine	Commentaire
stamina	entier 32 bits non négatif	Vigueur, résistance du civil. Valeur constante dans cette version du simulateur. Dans une version future, elle diminuera lorsque l'agent exécutera des actions, et il ne pourra plus rien faire lorsqu'elle vaudra 0. Le noyau la réaugmentera d'une valeur fixe à chaque cycle.
hp (*hit point*)	entier 32 bits non négatif	Cette valeur représente les «points de vie» du civil. Elle diminuera de la valeur contenue dans la propriété *damage* à chaque cycle. La valeur 0 signifie la mort.
damage	entier 32 bits non négatif	Cette propriété représente la gravité des atteintes à l'intégrité du civil (blessures, brûlures). La valeur peut être diminuée par un traitement médical.
buriedness	entier 32 bits non négatif	Cette propriété représente la profondeur à laquelle le civil est enfoui dans le bâtiment effondré. La valeur est le nombre de personnes (agents) nécessaires pour l'extraire des décombres. Une valeur supérieure à 1 signifie qu'il ne peut plus se déplacer lui-même. La valeur initiale est 0, et le simulateur de tremblement de terre calculera la mise à jour.
direction	entier 32 bits : $0 \sim 1\,295\,999$	Orientation du civil. La direction Nord a la valeur 0, et elle s'étend jusque $360*60*60-1 = 1\,295\,999$ secondes, dans le sens des aiguilles d'une montre. Cette valeur est notamment utilisée pour déterminer la direction des jets d'eau.

Propriété	Domaine	Commentaire
positionHistory	une liste d'ID	Liste d'IDs, tels que ceux de maisons ou de routes, que le civil a parcouru durant les cycles précédents. L'ordre est chronologique.

A.3 Voiture (agent)

Les voitures sont conduites par des agents civils. Leurs propriétés sont les mêmes que celles des agents civils.

A.4 Brigade de pompiers (agent)

Les brigades de pompiers ont les propriétés suivantes en plus des propriétés des agents civils :

Propriété	Domaine	Commentaire
waterQuantity	entier 32 bits non négatif	Représente la quantité d'eau du réservoir. Cette propriété n'est pas utilisée dans la version actuelle du simulateur. Son unité n'est pas spécifiée.
stretchedLength	entier 32 bits non négatif	Représente la longueur de tuyau entre l'agent et la bouche d'eau. Cette propriété n'est pas utilisée dans la version actuelle du simulateur. Son unité n'est pas spécifiée.

A.5 Equipe paramédicale (agent)

L'équipe paramédicale a les mêmes propriétés qu'un agent civil.

A.6 Force de police (agent)

Les forces de police ont les mêmes propriétés que les agents civils. Ils déblayent les routes pour permettre aux voitures, aux civils ou aux autres agents sauveteurs de passer.

A.7 Route (morceau du réseau routier)

Les routes sont représentées sur un graphe où les arrêtes sont des morceaux de routes et les noeuds des carrefours (intersection entre deux routes ou entre une route et l'entrée d'un bâtiment).

Propriété	Domaine	Commentaire
head	ID du noeud ou du bâtiment	ID du point extrémité initiale de la route.
tail	ID du noeud ou du bâtiment	ID du point extrémité finale de la route.
length	entier 32 bits non négatif	La longueur de la route. L'unité de mesure est le millimètre.
roadKind	entier 32 bits non négatif	La valeur indique le type de la route. Les valeurs 0x01, 0x02, 0x04 et 0x08 signifient respectivement «route surélevée», «pont», «tunnel» et «voie prioritaire».
carsPassToHead	entier 32 bits non négatif	Le nombre de voitures qui sont passées par le point extrémité initiale lors du cycle précédent. Il n'est pas utilisé dans cette version du simulateur.
carsPassToTail	entier 32 bits non négatif	Le nombre de voitures qui sont passées par le point extrémité finale lors du cycle précédent. Il n'est pas utilisé dans cette version du simulateur.
humanPassToHead	entier 32 bits non négatif	Le nombre de civils qui sont passés par le point extrémité initiale lors du cycle précédent.
humanPassToTail	entier 32 bits non négatif	Le nombre de civils qui sont passés par le point extrémité finale lors du cycle précédent.

Propriété	Domaine	Commentaire
width	entier 32 bits non négatif	La largeur de la route. L'unité de mesure est le millimètre.
block	entier 32 bits non négatif	La largeur de la partie de la route où les voitures ou les civils ne peuvent pas passer, dû aux effondrement de bâtiments, aux débris, etc. L'unité de mesure est le millimètre.
repairCost	entier 32 bits non négatif	Cette propriété représente le nombre de personnes (agents) nécessaires pour déblayer totalement la route.
medianStrip	0 ou 1	La valeur sera 1 lorsque la route sera pourvue d'une berme centrale, 0 sinon.
linesToHead	entier 32 bits non négatif	Le nombre de voies de circulation dans le sens extrémité finale vers extrémité initiale.
linesToTail	entier 32 bits non négatif	Le nombre de voies de circulation dans le sens extrémité initiale vers extrémité finale.
widthForWalkers	entier 32 bits non négatif	La largeur des trottoirs.

A.8 Noeud (carrefour, entrée, etc.)

Propriété	Domaine	Commentaire
x	entier 32 bits non négatif	Coordonnée x du noeud, en millimètre, à partir de l'origine.
y	entier 32 bits non négatif	Coordonnée y du noeud, en millimètre, à partir de l'origine.
edges	IDs de routes ou de bâtiments	Ensemble d'ID des objets, routes ou bâtiments, connectés au noeud.
signal	0 ou 1	La valeur est 1 s'il y a des signaux lumineux (feu rouge).

Propriété	Domaine	Commentaire
shortcutToTurn	liste d'entiers 32 bits non négatif	Liste du nombre des voies permettant de tourner à droite (gauche) sans passer par le carrefour, dans le cas d'un système routier droitier (gaucher). Les nombres sont dans l'ordre de la propriété *edges*.
pocketToTurnAcross	liste d'entiers 32 bits non négatif	Liste du nombre des voies aux carrefours, et de leur longueur, permettant de tourner à gauche (droite), dans le cas d'un système routier droitier (gaucher). Les nombres sont dans l'ordre de la propriété *edges*.
signalTiming	liste d'entiers 32 bits non négatif	Liste des temps de périodes des signaux lumineux pour chaque route de la propriété *edges*.

A.9 Rivière (morceau du réseau maritime)

Propriété	Domaine	Commentaire
head	ID d'un noeud du réseau maritime	ID du point extrémité initial de la rivière.
tail	ID d'un noeud du réseau maritime	ID du point extrémité final de la rivière.
length	entier 32 bits non négatif	Longueur de la rivière. L'unité de mesure est le millimètre.

A.10 Noeud de rivière (point de confluence du réseau maritime)

Propriété	Domaine	Commentaire
x	entier 32 bits non négatif	Coordonnée x du noeud.
y	entier 32 bits non négatif	Coordonnée y du noeud.
edges	IDs des rivières	Ensemble d'ID des objets connectés à ce noeud de rivière.

A.11 Bâtiment

Propriété	Domaine	Commentaire
x	entier 32 bits non négatif	Coordonnée x du centre du bâtiment.
y	entier 32 bits non négatif	Coordonnée y du centre du bâtiment.
floors	entier 32 bits non négatif	Le nombre d'étages du bâtiment.
buildingAttributes	entier : 0, 1 ou 2	La valeur représente le type de construction. Les valeurs 0, 1 et 2 signifient respectivement bâtiment «en bois», «avec trame en acier» et «en béton armé».
brokenness	entier : $0 \sim 100$	Valeur représentant le degré d'effondrement. Les valeurs vont de 0 (non effondré) à 100% d'effondrement.
ignition	entier : 0 ou 1	Cette valeur spécifie s'il y a, à l'initiation de la simulation, un foyer d'incendie dans le bâtiment (1) ou non (0). Les agents sauveteurs n'ont pas accès à cette information.

Propriété	Domaine	Commentaire
fieryness	entier : $0 \sim 7$	Valeur représentant l'intensité de l'incendie. Les valeurs 0, 1, 2, 3, 5, 6 et 7 représentent respectivement «non incendié», «début d'incendie», «incendie moyen», «incendie violent», «début d'incendie éteint», «éteint et moyennement consumé» et «entièrement consumé». La valeur 4 n'est pas utilisée.
entrance	liste d'IDs	Liste des IDs des objets connectés à l'entrée du bâtiment.
buildingShapeID	entier 32 bits non négatif	Nombre identifiant la forme du bâtiment.
buildingCode	entier 32 bits non négatif	Valeur associée à la structure du bâtiment.
buildingAreaGround	entier 32 bits non négatif	Surface du rez-de-chaussée.
buildingAreaTotal	entier 32 bits non négatif	Surface totale des étages.
buildingApexes	liste de coordonnées	Liste des coordonnées des sommets du bâtiment.

A.12 Bâtiments particuliers

Voici une série de bâtiments particuliers. Ils ont les mêmes propriétés que les objets bâtiments, mais méritent d'en être distingués de par leur spécificité.

- Le **refuge** : c'est un type de bâtiment dans lequel les civils peuvent être en sécurité, et trouver refuge.
- La **Caserne de pompiers** : c'est un bâtiment qui peut agir de manière autonome, dans le sens où les personnes qui l'occupent peuvent par exemple envoyer des informations aux pompiers sur le terrain.
- Le **Centre d'ambulances** : c'est un bâtiment qui peut agir de manière autonome, dans le sens où les personnes qui l'occupent peuvent par exemple envoyer des informations aux équipes paramédicales présentes sur le terrain.

- Le **Commissariat de police** : c'est un bâtiment qui peut agir de manière autonome, dans le sens où les personnes qui l'occupent peuvent par exemple envoyer des informations aux forces de police présentes sur le terrain.

Annexe B

Classement du championnat de 2001

Cette annexe contient les résultats du championnat de la RoboCupRescue de 2001. Les différentes situations utilisées, traces des simulations et code des agents sont disponibles sur la page internet [Report, 2001] d'où sont tirées les informations de cette annexe.

B.1 Equipes participantes

No.	Nom de l'équipe	Membres de l'équipe	Affilliation (Pays)
1	YabAI	Takeshi Morimoto	University of Electro-Communications (Japon)
2	Gemini-R	Masayuki Ohta	Tokyo Institute of Technology (Japon)
3	Rescue-ISI-JAIST	Takayuki Ito, Milind Tambe, Ranjit Nair, Stacy Marsella	Information Science Institute, University of Southern California (USA); Japan Advanced Institute of Science and Technology (Japon)
4	Arian	Jafar Habibi, Mazda Ahmadi, Ali Nouri	Sharif University of Technology (Iran)
5	JaistR	Kosuke Shinoda	Japan Advanced Institute of Sience and Technology (Japon)

6	NITRescue	Tetsuya Ezaki, Taku Sakushima, Nobuhiro Ito, Yoshiki Asai	Nagoya Institute of Technology (Japon)
7	RMIT-ON-FIRE	Lin Padgham, James Harland, John Thangarajah, Naveen Ruwanpura, Chandaka Fernando	Royal Melbourne Institute of Technology University (Australie)

B.2 Résultats

B.2.1 Eliminatoires

Les éliminatoires se sont déroulées sur la période du 4 au 6 août 2001.

Equipe no.	Points (Résultats de l'évaluation[1] (V))							Total	Position
	Situation[2]								
	1	2	3	4	5	6	7		
1	7 (18.25)	6 (19.36)	7 (29.38)	7 (41.59)	4 (28.55)	2 (30.48)	6 (13.23)	39	2
2	3 (56.89)	5 (21.46)	3 (45.67)	3 (56.80)	3 (51.85)	4 (25.56)	5 (15.24)	26	5
3	5 (30.70)	2 (27.75)	4 (40.55)	5 (43.76)	6 (22.30)	6 (24.44)	2.5 (17.22)	30.5	3
4	6 (30.37)	7 (17.27)	6 (33.44)	4 (54.71)	7 (22.30)	3 (27.36)	7 (10.24)	40	1
5	4 (30.70)	3 (22.81)	5 (38.55)	1 (62.84)	5 (22.31)	7 (24.41)	4 (17.22)	29	4
6	2 (58.89)	4 (22.46)	2 (66.88)	6 (43.75)	2 (54.86)	1 (36.79)	2.5 (17.22)	19.5	6
7	1 (60.91)	1 (36.85)	1 (35.63)	2 (62.84)	1 (56.86)	5 (24.74)	1 (17.50)	12	7

Dans l'ordre, les équipes qualifiées pour les quarts de finale sont : Arian, YabAI, Rescue-ISI-JAIST et JaistR. Notons que l'équipe RMIT-ON-FIRE n'avait développé que les brigades de pompiers.

[1]Pour rappel, $V = nombre\ de\ morts + \frac{I*D}{(B_{init}*H_{init})}$.

[2]Pour les éliminatoires, chaque équipe prit part aux 7 situations initiales préparées par chacune des différentes équipes.

B.2.2 Demi-finales

Demi Finale A (9 août 2001)		
Nom de l'équipe	Evaluation (V)	
YabAI	17.2792331741	20.2402455260
Rescue-ISI-JAIST	17.6728162788	20.4167472486

L'équipe gagnante de la première demi-finale est YabAI.

Semi Finale B (9 août 2001)			
Nom de l'équipe	Evaluation (V)		
Arian	13.2451824319	20.2565185571	19.2300132388
JaistR	14.6632544855	16.3553287093	17.3933319876

L'équipe JaistR fut avantagée lors des deux dernières évaluations, car les agents civils sortaient mystérieusement des bâtiments effondrés lors des premières minutes de la simulation. De plus, l'équipe JaistR a été disqualifiée car son code ne respectait pas les règles imposées par le championnat. L'équipe Arian fut donc déclarée vainqueur de la deuxième demi-finale.

B.2.3 Finale

Finale (10 août 2001)		
Nom de l'équipe	Evaluation (V)	
YabAI	19.2646659794	56.8223911629
Arian	33.3954412371	56.8551949182

YabAI est déclarée vainqueur du championnat «simulation» de la RoboCup-Rescue.

Annexe C

Spécification du code YabAI

Auteur du code : Morimoto Takeshi
(morimoto@takeopen.cs.uec.ac.jp)
Version du code présenté : 0.41 (pour le simulateur 0.31)
Adresse internet :
http://ne.cs.uec.ac.jp/~morimoto/rescue/ver0_41/index.html

Sommaire

C.1 Introduction

YabAI est le programme multiagent qui finit premier dans le cadre de la compétition RoboCupRescue qui a eu lieu à Seattle, en 2001. C'est un programme composé de plus de 5 000 lignes de code, réparties dans quelque 24 fichiers et 64 classes. Il a été écrit par Morimoto Takeshi, un étudiant japonais de l'université d'Electro-communications de Chofu à Tokyo (Japon), et commenté dans sa langue d'origine. C'est la difficulté que peut présenter l'analyse et la compréhension d'un pareil travail qui nous a amené à rédiger le présent manuel.

Cette annexe a pour objet la spécification du code constituant le système multiagent champion de la RoboCupRescue 2001 : YabAI. Ce manuel a été rédigé dans le but d'aider le programmeur qui voudrait l'utiliser pour construire son propre système multiagent (par exemple pour éviter d'avoir à réimplémenter le protocole de communication). Il peut aussi bien servir de manuel de référence des méthodes offertes que de document d'aide à la compréhension des algorithmes développés. Ce document a été rédigé en même temps qu'une lecture approfondie du code[1], avec parfois l'aide de M. Takeshi. Bien que ce document technique n'ait pas la prétention d'être parfait, il devrait être largement suffisant pour permettre au programmeur de construire son système multiagent en faisant abstraction des détails d'implémentation, afin de lui permettre de se concentrer uniquement sur l'élaboration de son architecture multiagent au travers des défis proposés par la RoboCupRescue.

Cette annexe est organisée à la manière des documents de spécification générés automatiquement par l'outil *javadoc*. Les premières pages listent les classes utilisées, les décrivent très brièvement et en montrent l'encapsulation. Ensuite chaque chapitre sera dédié à une de ces classes[2] et pratiquement toujours découpé de la même manière. Il sera habituellement introduit par une petite explication du but de la classe. Il présentera alors, si la classe en contient, les champs (constantes, variables...), les constructeurs (hormis le constructeur habituel dont toute classe java dispose), et enfin les diverses méthodes qui la composent. Il arrivera pour certaines classes exceptionnelles que cette structure change légèrement, mais cela ne devrait pas incommoder le lecteur.

[1] Certains «bogues» ont d'ailleurs été relevés. Ils sont signalés dans les chapitres auxquels ils appartiennent et dans la plupart des cas, nous en proposerons une correction.

[2] Exception faite de la classe KdTree.

C.2 Liste des fichiers

Vous trouverez ci-dessous la liste des fichiers qui implémentent les agents YabAI. Elle est donnée dans l'ordre de lecture conseillé par M. Takeshi, et avec ses commentaires traduits en français. Nous ne suivrons pas exactement cet ordre, considérant par exemple que Constant.java devrait venir beaucoup plus tôt puisque c'est la classe dont pratiquement toutes les autres découlent. Nous resterons toutefois fort proche de cette liste.

1. **Predicate.java** évalue un objet.
2. **Function.java** accède aux propriétés d'un objet.
3. **RescueObject.java** définit tous les objets dans l'espace du désastre.
4. **Domain.java** est un ensemble fonctionnel.

5. **ObjectPool.java** modélise l'image du monde d'un agent en particulier.

6. **CostFunction.java** est utilisée pour déterminer des itinéraires.
7. **Route.java** est un itinéraire de déplacement.
8. **Router.java** détermine les itinéraires de déplacement.

9. **IO.java** communique entre le système et les agents.

10. **Main.java** construit et démarre les agents.

11. **Message.java** représente les messages parlés par un agent.
12. **Action.java** représente les action exécutées par un agent.
13. **History.java** représente l'historique des actions d'un agent.

14. **Controller.java** contrôle l'agent.

15. **Constants.java** définit les constantes.
16. **CivilianController.java** contrôle un agent civil.

17. **KdTree.java** accède à un ensemble d'objets d'une région.
18. **FireBrigadeController.java** contrôle un agent brigade de pompiers.
19. **AmbulanceTeamController.java** contrôle un agent équipe paramédicale.
20. **PoliceForceController.java** contrôle une agent force de police

21. **FireStationController.java** contrôle un agent caserne de pompiers.
22. **AmbulanceCenterController.java** contrôle un agent centre paramédical.
23. **PoliceOfficeController.java** contrôle une agent commissariat de police.

24. **Scope.java** visualise le monde modélisé.

C.3 Hiérarchie des classes

- class **rescue.Predicate**
  - class EqualP
  - class ContainP
  - class EvalPositionP
  - class EvalLocationP
- class **rescue.Function**
  - class RouteCostF
  - class PriorityF
  - class DistanceF
- class **rescue.Constants**
- class **rescue.RescueObject**
  - class VirtualObject
    - ⋆ class World
  - class RealObject
    - ⋆ class MotionlessObject
      - ⊙ class DummyObject
      - ⊙ class PointObject
        - class Building
          - ⟶ class FireStation
          - ⟶ class AmbulanceOffice
          - ⟶ class PoliceOffice
          - ⟶ class Refuge
        - class Vertex
          - ⟶ class Node
          - ⟶ class RiverNode
      - ⊙ class Edge
        - class Road
        - class River
    - ⋆ class MovingObject
      - ⊙ class Humanoid
        - class Civilian
        - class Car
        - class FireBrigade
        - class AmbulanceTeam
        - class PoliceForce
- class **rescue.Domain**
- class **rescue.ObjectPool**
- class **rescue.CostFunction**
- class **rescue.Route**

- class **rescue.Router**
- class **rescue.IO**
- class **rescue.Main**
- class **rescue.Message**
- class **rescue.Action**
  - class RestAction
  - class UnloadAction
  - class RescueAction
  - class LoadAction
  - class ClearAction
  - class MoveAction
  - class TellAction
  - class SayAction
  - class ExtinguishAction
- class **rescue.History**
- class **rescue.KdTree**
- class **rescue.Controller**
  - class CivilianController
  - class FireBrigadeController
  - class FireStationController
  - class PoliceForceController
  - class PoliceOfficeController
  - class AmbulanceTeamController
  - class AmbulanceCenterController
- class **rescue.scope**

C.4 Abstract Class Predicate

Cette classe abstraite représente la logique des prédicats. On y retrouve les prédicats ET, OU, NON, et => (implique), ainsi qu'une fonction d'évaluation abstraite (Pour rappel, l'évaluation de (x_1 ET x_2) donne (l'évaluation de x_1) «et» (l'évaluation de x_2). Le «et» est le méta-opérateur logique). Des exemples d'utilisation sont donnés au point C.4.6 page 123.

C.4.1 Méthodes

- ⋆ `abstract boolean` **eval** `(Object obj)`
  Fonction d'évaluation abstraite d'objets.
- ⋆ `Predicate` **and**`(Predicate c)`
  Prédicat «et».
- ⋆ `Predicate` **or**`(Predicate c)`
  Prédicat «ou».
- ⋆ `Predicate` **not**`()`
  Prédicat «non».
- ⋆ `Predicate` **implies**`(Predicate c)`
  Prédicat «implique».

C.4.2 Sous-classes de Predicate

Les sous-classes EqualP et ContainP implémentent l'évaluation de l'égalité ainsi que l'évaluation de l'inclusion ensembliste, tandis que les sous-classes EvalPositionP et EvalLocationP servent à évaluer la position (pour un objet mobile) ou la localisation[3] d'un objet à partir d'un prédicat particulier, donné en paramètre. Vu la similitude de ces 2 dernières, seule la première des deux sera spécifiée.

C.4.3 Class EqualP

Classe qui implémente l'égalité.

Champ

- `Object m_obj`
  Objet de comparaison pour l'égalité.

[3]Notons dès à présent la différence entre position et localisation par un simple exemple : un agent civil situé dans une ambulance aura comme localisation la valeur de l'identifiant de la route sur laquelle est l'ambulance, et comme position la valeur de l'identifiant de l'ambulance elle même.

Constructeur

⊙ **EqualP**(`Object obj`)
Le constructeur affecte au champ `m_obj` l'objet `obj`.

Méthodes

⋆ `void` **setObject**(`Object obj`)
Affecte au champ `m_obj` l'objet `obj`.

⋆ `boolean` **eval**(`Object obj`)
Renvoie (`obj==m_obj`).

C.4.4 Class ContainP

Classe qui implémente l'inclusion ensembliste.

Champ

• `HashSet m_set`
Ensemble servant à vérifier l'inclusion.

Constructeur

⊙ **ContainP**(`HashSet set`)
Le constructeur affecte au champ `m_set` l'ensemble `set`.

Méthode

⋆ `boolean` **eval**(`Object obj`)
Renvoie `true` si `obj` est contenu dans l'ensemble `m_set`.

C.4.5 Class EvalPositionP (similaire à EvalLocationP)

Classe qui implémente l'évaluation de la position d'un objet de la classe MovingObject par rapport à un prédicat donné. Un exemple est donné en C.4.6 page 123.

Champ

• `Predicate m_cond`
Champ contenant la condition qui va servir à vérifier la position de l'objet mobile.

Constructeur

⊙ **EvalPositionP**(`Predicate cond`)
Le constructeur affecte au champ `m_cond` le prédicat `cond`.

Méthodes

- ⋆ `void` **setPositionCondition**(`Predicate cond`)
  Affecte au champ `m_cond` le prédicat cond.
- ⋆ `boolean` **eval**(`Object obj`)
  Applique la fonction d'évaluation du prédicat `m_cond` à `obj.position()`

C.4.6 Exemples

Nous avons maintenant assez d'outils pour présenter un premier exemple dont le but est de clarifier la compréhension de la classe Predicate. Nous présentons ici un code qui définit un prédicat permettant, à partir d'un objet donné, d'évaluer si au moins un des deux objets `obj1` et `obj2` lui est égal :

```
Predicate P1 = new EqualP(obj1) ;
Predicate P2 = new EqualP(obj2) ;
Predicate egal_a_P1_ou_a_P2 = P1.or(P2)
```

Pour récupérer le résultat :

```
boolean resultat = egal_a_P1_ou_a_P2.eval(objet_donné) ;
```

Voici un deuxième exemple permettant de comprendre la méthode EvalPositionP(Predicate c). Il définit un prédicat qui permet de savoir si l'agent ambulance1 est à la position du civil blessé pour le soigner :

```
Predicate P1 = new EqualP(civil.position()) ;
Predicate peut_soigner_civil = new EvalPositionP(P1) ;
```

Pour récupérer le résultat :

```
boolean soigne = peut_soigner_civil.eval(ambulance1) ;
```

En voici un dernier exemple qui utilise la classe des fonctions et celle des constantes (Voir chapitre C.5 et C.6), qui nous donne un prédicat qui peut évaluer si un bâtiment est en train de brûler (il faut juste savoir que `FIERYNESS_F.gte(1)` est un prédicat dont l'évaluation nous dit si une incendie s'est déclaré dans l'objet évalué, et `FIERYNESS_F.lte(3)` un prédicat dont l'évaluation permet de savoir que soit l'objet n'a pas eu d'incendie soit il n'est pas encore éteint) :

```
Predicate burning = FIERYNESS_F.gte(1).and(FIERYNESS_F.lte(3))
```

Pour récupérer le résultat avec n'importe quel bâtiment `bldg` :

```
boolean is_burning = burning.eval(bldg)
```

C.5 Abstract Class Function

Cette classe, un peu à la manière de la classe Predicate, représente l'algèbre de base. On y retrouve les fonctions plus, moins, fois, divisé, ainsi qu'une définition des équations et inéquations.

C.5.1 Méthodes

- ⋆ `abstract int` **eval**`(Object obj)`
  Fonction d'évaluation abstraite d'objets. Notez qu'elle renvoie ici un entier et non un booléen comme dans les prédicats. Des exemples sont présentés au point suivant.
- ⋆ `Function` **add**`(Function f)` ; `Function` **add**`(int n)`
  Fonctions d'additions.
- ⋆ `Function` **sub**`(Function f)` ; `Function` **sub**`(int n)`
  Fonction de soustraction.
- ⋆ `Function` **mul**`(Function f)` ; `Function` **mul**`(int n)`
  Fonction de multiplication.
- ⋆ `Function` **div**`(Function f)` ; `Function` **div**`(int n)`
  Fonction de division (réelle).
- ⋆ `Predicate` **eq**`(Function seuil)` ; `Predicate` **eq**`(int seuil)`
  Renvoie le prédicat correspondant à l'égalité (une équation).
- ⋆ `Predicate` **gt**`(Function seuil)` ; `Predicate` **gt**`(int seuil)`
  Renvoie le prédicat *Greater Than* («plus grand que» : prédicat $>$).
- ⋆ `Predicate` **lt**`(Function seuil)` ; `Predicate` **lt**`(int seuil)`
  Renvoie le prédicat *Lower Than* («plus petit que» : prédicat $<$).
- ⋆ `Predicate` **gte**`(Function seuil)` ; `Predicate` **gte**`(int seuil)`
  Renvoie le prédicat *Greater Than or Equal* ($\geq$).
- ⋆ `Predicate` **lte**`(Function seuil)` ; `Predicate` **lte**`(int seuil)`
  Renvoie le prédicat *Lower Than or Equal* ($\leq$).

C.5.2 Exemple

Voici comment coder l'inéquation $x * 5 + 7 \leq 9$:

```
Function f = new Function() ; //x
Function f2 = f.mul(5) ; //x * 5
Function f3 = f2.add(7) ; //x * 5 + 7
Predicate P = f3.lte(9) ; //x * 5 + 7 ≤ 9
```

C.6 Interface Constants

Cette interface contient toutes les constantes définies dans le manuel du simulateur (entêtes des paquets échangés, constantes associées aux types d'objets, à leurs propriétés) ainsi que diverses autres utilisées de diverses manières. Outre ces constantes, elle définit aussi tout un jeu de fonctions et de prédicats (dont la fonction d'évaluation est implémentée) permettant de manipuler les objets du monde. Enfin, il est important de noter que c'est sur cette classe que tout le reste repose, puisque la quasi totalité des classes qui suivent implémentent celle-ci. Une dernière précision : tous ces champs sont «final» (valeurs définitives).

C.6.1 Champs des constantes (1)

HEADER_NULL	AK_CONNECT	AK_ACKNOWLEDGE
KA_CONNECT_OK	KA_CONNECT_ERROR	KA_SENSE
KA_HEAR	SK_CONNECT	SK_ACKNOWLEDGE
SK_UPDATE	KS_CONNECT_OK	KS_CONNECT_ERROR
KS_COMMAND	KS_UPDATE	VK_CONNECT
VK_ACKNOWLEDGE	KV_CONNECT_OK	KV_CONNECT_ERROR
KV_UPDATE	GK_CONNECT_OK	GK_CONNECT_ERROR
KG_CONNECT	KG_ACKNOWLEDGE	KG_UPDATE
HEARER_COMMAND_MIN		

Tous ces champs contiennent les valeurs (int) nécessaires pour garnir les entêtes des messages échangés entre les différents modules.

TYPE_NULL	TYPE_ROAD	TYPE_RIVER
TYPE_BUILDING	TYPE_REFUGE	TYPE_FIRE_STATION
TYPE_AMBULANCE_CENTER	TYPE_POLICE_OFFICE	TYPE_RIVER_NODE
TYPE_NODE	TYPE_WORLD	TYPE_CIVILIAN
TYPE_FIRE_BRIGADE	TYPE_AMBULANCE_TEAM	TYPE_POLICE_FORCE
TYPE_CAR	TYPE_DUMMY	

Tous ces champs contiennent les valeurs (int) correspondantes aux types d'objets qu'ils représentent.

PROPERTY_NULL	PROPERTY_WIDTH
PROPERTY_BUILDING_AREA_TOTAL	PROPERTY_START_TIME
PROPERTY_SIGNAL_TIMING	PROPERTY_LINES_TO_HEAD
PROPERTY_POSITION_HISTORY	PROPERTY_CARS_PASS_TO_HEAD
PROPERTY_LATITUDE	PROPERTY_DIRECTION
PROPERTY_BLOCK	PROPERTY_STAMINA

PROPERTY_HEAD	PROPERTY_EDGES
PROPERTY_BUILDING_CODE	PROPERTY_SIGNAL
PROPERTY_FIERYNESS	PROPERTY_WIND_DIRECTION
PROPERTY_LENGTH	PROPERTY_FLOORS
PROPERTY_HUMANS_PASS_TO_TAIL	PROPERTY_ROAD_KIND
PROPERTY_REPAIR_COST	PROPERTY_IGNITION
PROPERTY_BUILDING_AREA_GROUND	PROPERTY_BROKENESS
PROPERTY_WIND_FORCE	PROPERTY_SHORTCUT_TO_TURN
PROPERTY_WIDTH_FOR_WALKERS	PROPERTY_MEDIAN_STRIP
PROPERTY_Y	PROPERTY_ENTRANCE
PROPERTY_X	PROPERTY_POSITION
PROPERTY_POSITION_EXTRA	PROPERTY_WATER_QUANTITY
PROPERTY_LONGITUDE	PROPERTY_STRETCHED_LENGTH
PROPERTY_HUMANS_PASS_TO_HEAD	PROPERTY_BURIEDNESS
PROPERTY_LINES_TO_TAIL	PROPERTY_HP
PROPERTY_BUILDING_APEXES	PROPERTY_DAMAGE
PROPERTY_CARS_PASS_TO_TAIL	PROPERTY_BUILDING_ATTRIBUTES
PROPERTY_POCKET_TO_TURN_ACROSS	PROPERTY_TAIL

Tous ces champs contiennent les valeurs (int) correspondantes aux propriétés qu'ils représentent.

AK_REST	AK_MOVE	AK_LOAD	AK_UNLOAD	AK_SAY
AK_TELL	AK_EXTINGUISH	AK_STRETCH	AK_RESCUE	AK_CLEAR

Ces champs contiennent les valeurs (int) utilisées pour communiquer au noyau les actions effectuées par les agents.

C.6.2 Champs des fonctions

Les descriptions qui suivent commencent souvent par «Renvoie...». Ceci est un abus de langage qui signifie «Si on évalue cette fonction avec la méthode `eval(RescueObject)`, elle renverra...».

- `Function ID_F` ; `Function TYPE_F`
  Ces fonctions renvoient l'ID ou le type (voir point C.6.1 page 125) de l'objet RescueObject évalué.
- `Function X_F` ; `Function Y_F`
  Renvoie la position x / y de l'objet RescueObject évalué.
- `Function START_TIME_F`
  Renvoie le (la valeur du) champ `m_startTime` de l'objet World évalué.
- `Function LONGITUDE_F` ; `Function LATITUDE_F`
  Renvoie la longitude / latitude de l'objet World évalué.

- `Function WIND_FORCE_F` ; `Function WIND_DIRECTION_F`
  Renvoie la force / direction du vent, de l'objet World évalué.
- `Function POSITION_EXTRA_F`
  Renvoie le (la valeur du) champ `m_positionExtra` d'un MovingObject.
- `Function STAMINA_F` ; `Function HP_F`
  Renvoie le champ `m_stamina` / `m_hp` d'un objet Humanoid.
- `Function DAMAGE_F` ; `Function BURIEDNESS_F`
  Renvoie le champ `m_damage` / `m_buriedness` d'un objet Humanoid.
- `Function FIERYNESS_F` ; `Function BROKENNESS_F`
  Renvoie le champ `m_fieryness` / `m_brokeness` d'un objet Building.
- `Function BUILDING_AREA_TOTAL_F` ; `Function BUILDING_CODE_F`
  Renvoie `m_buildingAreaTotal` / `m_buildingCode` d'un objet Building.
- `Function WATER_QUANTITY_F` ; `Function STRETCHED_LENGTH_F`
  Renvoie `m_waterQuantity` / `m_stretcherLength` d'un objet FireBrigade.
- `Function LENGTH_F`
  Renvoie `m_length` d'un objet Edge (arrête du graphe).
- `Function WIDTH_F` ; `Function BLOCK_F`
  Revoit `m_width` / `m_block` d'un objet Road.
- `Function REPAIR_COST_F` ; `Function BLOCKED_LINES_F`
  Renvoie `m_repairCost` / le nombre de voies obstruées, d'un objet Road.
- `Function LINES_TO_HEAD_F` ; `Function LINES_TO_TAIL_F`
  Renvoie le nombre de voies de circulation de la route dans le sens tête-queue / queue-tête, ou encore le champ `m_linesToHead` / `m_linesToTail` d'un objet Road.
- `Function ALIVE_LINES_TO_HEAD_F` ;
  `Function ALIVE_LINES_TO_TAIL_F`
  Renvoie le nombre de voies non obstruées dans le sens tête-queue / queue-tête d'un objet Road.
- `Function TIME_UPDATES_F`
  Renvoie `m_time` d'un RescueObject (temps de la dernière mise à jour).
- `Function LIFESPAN_F`
  Cette fonction permet d'évaluer la durée de vie d'un Humanoid. Elle renverra 10 000 + son nombre de points de vie (`m_hp`) pour un humanoïde sain (donc 20 000 au maximum), 0 pour un mort, et `m_hp` divisé par la valeur représentant la gravité de ses besoins en soins (`m_damage`) dans les autres cas. Ceci n'équivaut pas exactement au nombre de cycles qu'il reste à vivre au civil. C'est en fait une borne supérieure, car le champ `m_damage` est variablement croissant au cours du temps.

- Function `MSG_PREDICATE_F`
  Renvoie la valeur entière représentée par l'objet `String`. Les messages (Voir C.16, page 160) sont composés de 2 entiers qui se suivent ; un prédicat et l'ID d'un objet. Et ces entiers sont représentés par des chaînes de caractères comme par exemple "4 12345". Notez que `MSG_CLEAR_ROAD` est 4 (Voir point C.6.4). La fonction `MSG_PREDICATE_F` renvoie le prédicat du message, c'est-à-dire 4, ou `MSG_CLEAR_ROAD`, dans cet exemple.

C.6.3 Champs des prédicats (1)

- `Predicate TRUE_P` ; `Predicate FALSE_P`
  L'évaluation de ce prédicat renvoie toujours `true` / `false`.
- `Predicate CIVILIAN_P` ; `Predicate FIRE_BRIGADE_P` ;
  `Predicate AMBULANCE_TEAM_P` ; `Predicate POLICE_FORCE_P` ;
  `Predicate ROAD_P` ; `Predicate NODE_P` ;
  `Predicate BUILDING_P` ; `Predicate REFUGE_P`
  Ce sont des prédicats dont l'évaluation renvoie `true` si l'objet est un objet Civilian, FireBrigade, AmbulanceTeam, PoliceForce, Road, Node, Building, ou Refuge, selon le prédicat.
- `Predicate SPECIAL_BUILDING`
  L'évaluation de ce prédicat donne `true` si l'objet est un des suivants : Refuge, FireStation, AmbulanceCenter, ou PoliceOffice.
- `Predicate BURNING_P` ; `Predicate NON_BURNING_P`
  L'évaluation de ce prédicat donne `true` / `false` si le champ `m_fieryness` de l'objet Building évalué est compris entre 1 et 3, c'est-à-dire si le bâtiment brûle.
- `Predicate EXTINGUISHED_OR_BURNED_OUT_P`
  L'évaluation de ce prédicat donne `true` si le champ `m_fieryness` de l'objet Building évalué est compris entre 5 et 7, c'est-à-dire si un incendie s'y est déclaré et a été éteint ou a entièrement consumé le bâtiment.

C.6.4 Champs des constantes (2)

`LOAD_FACTOR` | `DIV_LF`

Ces champs (`float`) contiennent des facteurs de multiplication utilisés pour calculer la taille de certains ensembles (ils interviennent notamment dans les constantes qui vont suivre ainsi que dans la construction des ensembles du point C.8.2 page 145).

SIZE_OF_POOL	SIZE_OF_ROAD
SIZE_OF_BUILDING	SIZE_OF_HUMANOID
SIZE_OF_REFUGE	SIZE_OF_CIVILIAN
SIZE_OF_FIREBRIGADE	SIZE_OF_POLICE_FORCE
SIZE_OF_AMBULANCE_TEAM	SIZE_OF_MISSED_HUMANOID
SIZE_OF_MISSED_CIVILIAN	SIZE_OF_MISSED_FIRE_BRIGADE
SIZE_OF_MISSED_AMBUMANCE_TEAM	SIZE_OF_MISSED_POLICE_FORCE
SIZE_OF_VISIBLE_BUILDING	

Ces champs (int) sont utilisés pour définir la taille des ensembles d'objets respectifs (ils sont notamment utilisés lors de construction d'objets Domain, au chapitre C.9 concernant les ensembles d'objets, page 147). D'après M. Takeshi, ils ont été introduits dans le but de gagner du temps et de la place mémoire. Cependant, il pense maintenant qu'ils n'ont pas de raison d'être et devraient être effacés dans une prochaine version.

IO_BUF_SIZE | IO_RECEIVE_OUT_TIME

Ces champs (int) définissent la taille des tampons d'échange (=1600), ainsi que le temps maximal permis lors des communications (=20 ms).

AGENT_TYPE_CIVILIAN	AGENT_TYPE_FIRE_BRIGADE
AGENT_TYPE_FIRE_STATION	AGENT_TYPE_AMBULANCE_TEAM
AGENT_TYPE_AMBULANCE_CENTER	AGENT_TYPE_POLICE_FORCE
AGENT_TYPE_POLICE_OFFICE	

Ce sont les valeurs (int) utilisées par les agents pour se connecter au noyau.

MSG_NONSENCE	MSG_PLEASE_CLEAR
MSG_CLEAR_ROAD	MSG_REPORT_CLEARED
MSG_THERE_IS_NO_INJ	MSG_INJ_IS_HERE
MSG_DANGEROUS_INJ_IS_HERE	MSG_SAFETY_INJ_IS_HERE
MSG_EASY_INJ_IS_HERE	

Ce sont les valeurs (int) associées aux messages que s'échangent les agents.

TIME_SIMULATION | TIME_PREPARING_SIMULATION

Ce sont des constantes (int) fixées pour déterminer le temps de la simulation (=300 tours) ainsi que le temps de préparation (pré-simulation, 2 tours).

C.6.5 Champs des prédicats (2)

- `Predicate HAVE_SEEN_P ; Predicate HAVE_NOT_SEEN_P`
  L'évaluation d'un objet par ce prédicat renvoie `true` / `false` si l'objet a été vu après l'initialisation du système (après la catastrophe).
- `Predicate MOVABLE_P ; Predicate UNMOVABLE_P`
  Renvoie `true` / `false` s'il existe au moins une voie de circulation non obstruée dans les 2 sens. Techniquement, `MOVABLE_P` est le prédicat `(ALIVES_LINES_TO_HEAD_F.gte(1)) .and(ALIVES_LINES_TO_TAIL_F .gte(1))`.
- `Predicate MOVABLE_EIGHTE_DIRECTION_P`
  Renvoie `true` si la route est praticable dans au moins un des deux sens. Techniquement, c'est le prédicat `(ALIVES_LINES_TO_HEAD_F.gte(1)) .or(ALIVES_LINES_TO_TAIL_F.gte(1))`.
- `Predicate UNMOVABLE_EIGHTE_DIRECTION_P`
  Renvoie `true` si un des deux sens de circulation est complètement obstrué. Techniquement, c'est le prédicat `(ALIVES_LINES_TO_HEAD_F .eq(0)).or(ALIVES_LINES_TO_TAIL_F.eq(0))`.
- `Predicate UNMOVABLE_BOTH_DIRECTION_P`
  Renvoie `true` si la rue est complètement obstruée dans les deux sens. Techniquement, c'est le prédicat `(ALIVES_LINES_TO_HEAD_F.eq(0)) .and(ALIVES_LINES_TO_TAIL_F.eq(0))`.
- `Predicate INJURIED_P`
  Renvoie `true` si l'objet subit des dommages, est encore en vie et n'est ni dans une ambulance, ni dans un refuge (d'après M. Takeshi, ce prédicat est bogué).
- `Predicate CLEARABLE_P ; Predicate UNCLEARABLE_P`
  Renvoie `true` / `false` s'il existe des voies de circulation bloquées (et donc «nettoyables»).

C.6.6 Champs des constantes (3)

Ces constantes-ci touchent divers domaines, et seules les plus importantes sont commentées.

`int CIV_UNACTABLE_DAMAGE`

C'est la valeur au dessous de laquelle un humanoïde ne peut plus agir. Si un humanoïde à son champ `m_damage` au dessus de cette valeur, ses actions ne sont plus prises en compte (Voir chapitre C.7.18 page 143).

`VISIBLE_DISTANCE`	`MAX_NUM_TALKING_A_TIME`	`MAX_NUM_HEARING_A_TIME`
`LIMIT_ROUTE_LENGTH`	`DAMAGE_NEED_TO_CARE`	

Ces valeurs (int) sont fixées par le comité. La distance visible est fixée à 10 000mm, le nombre de messages dits ou entendus est fixé à 4, et le seuil de dommage à prendre en compte est 1.

```
double PROBABILITY_GIVING_WAY | int  LENGTH_OF_GIVE_WAY
```

Ces constantes sont utilisées lorsqu'il y a des embouteillages. Il arrive alors que certains agents laissent passer d'autres agents. Elles sont utilisées dans la fonction **expand**(Predicate, Domain) de la classe Route (Chapitre C.11, page 153).

```
int TIME_FORGETING_UNMOVABLE
```

Les agents mémorisent les routes qui sont inutilisables. Cependant, après un certain temps, ils peuvent les oublier, considérant qu'elles ont sans doute été dégagées par la police.

```
int SPEED_OF_CAR
```

Vitesse des voitures : $16 * 60 * 1000 = 960\,000$ mm/min (environ 57.6km/h).

```
boolean AMB_DO_DANGEROUS_RESCUE   | int AMB_CRITICAL_DAMAGE
int AMB_SAFETY_HP                 | double AMB_SAFETY_LF_RATE
int AMB_SAFETY_LIFESPAN           | int AMB_DISTANCE_NEAR_BURNING
int EXTINGUISHABLE_DISTANCE       | int EXTINGUISHABLE_QUANTITY
int POSSIVELY_EXTINGUISHABLE_TIME | boolean DIVIDING_EXTINGUISH
int EXTINGUISHABLE_TIME_BY_MANY_FB|
```

Ces constantes sont utilisées par les ambulances : quels sont leurs seuils critiques de dommages (100), de points de vie (8 000), la distance à laquelle elles peuvent éteindre un incendie (30 000mm), etc.

```
double RATE_OF_MIKIRI
```

Constante inutilisée qui devra servir à estimer l'urgence d'un incendie.

```
int DISTANCE_OF_NEIGHBORHOOD      | int MUL_WORTH
int VARIANCE_MAX_WORTH            | int MAX_WORTH_UPPER
boolean SUPPORT_EARLY_EXTINGUISH  | int DISTANCE_PRIOR_RESCUE_HUMAN
boolean REPORT_CLEARING_ONE_WAY   | int UNCERTAIN_ROAD_REPAIR_COST
int WORTH_CLEARING_ROAD           | int TIME_WITHIN_REPORTED_PRI...
int NUM_REPORTED_PRIOR_CLEARING   | int WIDTH_CLEARED_BY_CLEARING
int TIME_ACTIVE_SUPPORT_EXTING... | boolean DO_PRE_INFO
String PRE_INFO_DIR_NAMME         | int TIME_SAVE_POWER_MSG
```

Constantes non commentées.

SCOPE_NUM_DOMAIN	SCOPE_DEST
SCOPE_ROAD	SCOPE_BLOCKED_ROAD
SCOPE_REPORTED_ROAD	SCOPE_BUILDING
SCOPE_ROUTE	SCOPE_INIT_CIV_BUIL
SCOPE_MISSED_CIVILIAN	SCOPE_MISSED_FIRE_BRIGADE
SCOPE_MISSED_AMBULANCE_TEAM	MISSED_POLICE_FORCE
SCOPE_CIVILIAN	SCOPE_FIRE_BRIGADE
SCOPE_AMBULANCE_TEAM	SCOPE_POLICE_FORCE
SCOPE_FIRE_TARGET	SCOPE_REPORTED_HUMANOID
USING_SCOPE	SCOPING_ID

Constantes (int sauf using_scope, boolean) utilisées par la classe Scope. Pour bénéficier du débogueur visuel, il faut fixer le champ USING_SCOPE à true, et le champ SCOPING_ID avec l'ID de l'agent à surveiller (pour plus de détails, voir point C.21, page 175).

USING_PRINT	USING_PRINT2	USING_PRINT_SHORT_STATE
MISC_CHECK	USING_DEBUG_PRINT	

Ces champs (boolean) sont utilisés pour le débogage.

C.7 Abstract Class RescueObjects

Cette classe abstraite est celle qui va permettre la modélisation du monde dans lequel vont évoluer catastrophes et agents. Chaque objet qui garnit le monde de la simulation (bâtiments, routes, rivières, civils, ...) héritera de cette classe maîtresse.

C.7.1 Hiérarchie

Avant toute chose et pour plus de clarté, voyons la hiérarchie de la classe RescueObject et de toutes celles qui vont en découler.

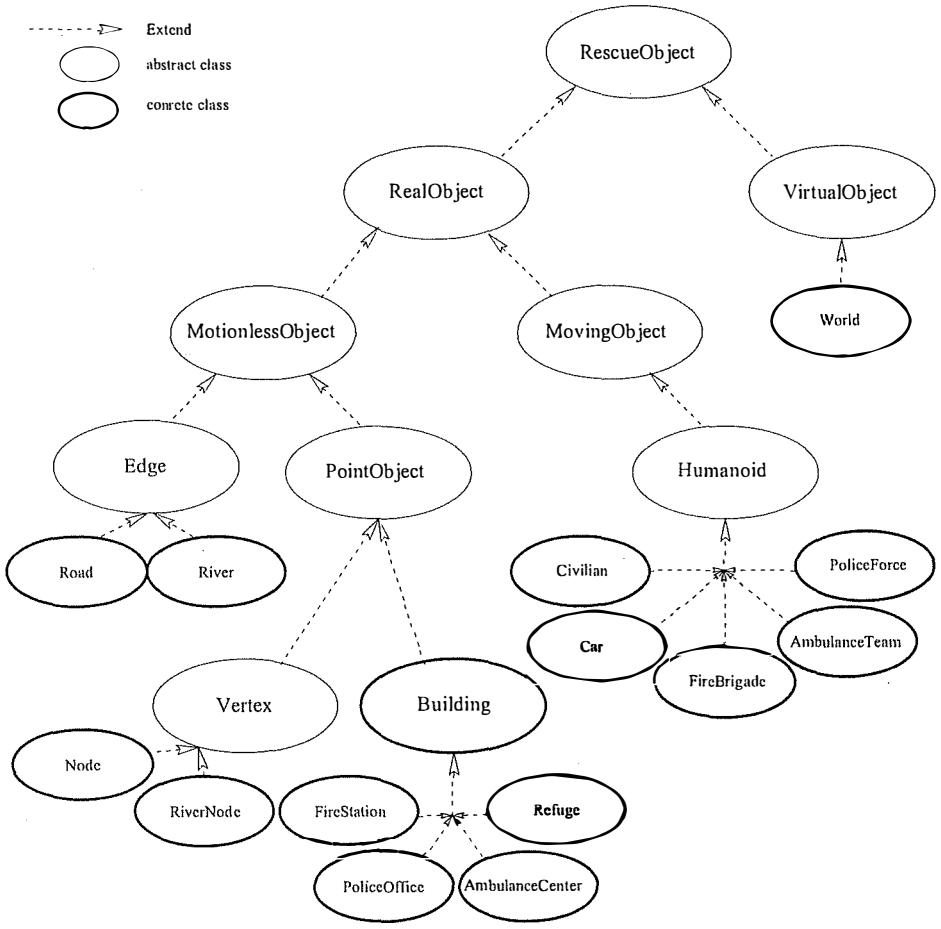


FIG. C.1 – Modélisation du monde pour les agents YabAI.

C.7.2 Champs

- `int m_id`
  Ce champ contient l'ID (numéro identifiant) de l'objet.
- `int m_time`
  Ce champ contient le temps auquel a eu lieu la dernière mise à jour.
- `ObjectPool m_myWorld`
  Ce champ contient le monde auquel appartient l'objet.
- `final DummyObject DUMMY_OBJECT`
  Objet factice. Utilisé parfois au point C.7.11 page 138.

C.7.3 Méthodes

- ⋆ `abstract int` **type**()
  Renvoie le type de l'objet (Voir C.6.1 page 125).
- ⋆ `abstract int` **x**(), `abstract int` **y**()
  Renvoie la position x / y de l'objet.
- ⋆ `abstract MotionlessObject` **location**()
  Renvoie la localisation de l'objet.
- ⋆ `int` **id**()
  Renvoie l'ID de l'objet contenu dans le champ `m_id`.
- ⋆ `int` **time**()
  Renvoie le temps auquel l'objet a été mis à jour la dernière fois (champ `m_time`).
- ⋆ `ObjectPool` **myWorld**()
  Renvoie le monde auquel appartient l'objet (champ `m_myWorld`).
- ⋆ `void` **setId**(`int id`)
  Assigne l'ID `id` à l'objet (au champ `m_id`).
- ⋆ `void` **setTime**(`int time`)
  Assigne le temps d'actualisation `time` à l'objet (au champ `m_time`).
- ⋆ `void` **setMyWorld**(`ObjectPool myWorld`)
  Assigne le monde MyWorld à l'objet (au champ `m_myWorld`).
- ⋆ `void` **input**(`int property, int[] value`)
  Le but de cette méthode est de garnir les champs de l'objet. A ce niveau, elle ne fait rien.
- ⋆ `String` **toString**()
  Renvoie la chaîne de caractère «id + (+ nom_de_l'objet + @ + code_de_hachage) @ + time» avec le + l'opérateur de concaténation.

C.7.4 Class VirtualObject et World

La classe abstraite VirtualObject, sous-classe de RescueObject, n'a aucun champ, ni aucune méthode. La classe World est la sous-classe concrète de la classe VirtualObject, que nous présentons ici.

Champs

- `int m_startTime`
  Contient la valeur du temps à la création du monde.
- `int m_longitude`, `int m_latitude`
  Contiennent la longitude et la latitude du centre de la carte.

- `int m_windForce, int m_windDirection`
  Contiennent la force et la direction du vent.

Méthodes

- ⋆ `int type(), int x(), int y()`
  Renvoie le type, la position x, ou y de l'objet.
- ⋆ `MotionlessObject location()`
  Renvoie `null`.
- ⋆ `int startTime(), int longitude(), int latitude(), int windForce(), int windDirection()`
  Renvoient les champs de même noms.
- ⋆ `void setLongitude(int value), void setLatitude(int value), void setWindForce(int value), void setWindDirection(int value), void setStartTime(int value)`
  Affecte les champs de même noms avec la valeur `value`.
- ⋆ `void input(int property), int[] value)`
  Affecte `value[0]` au champ correspondant à la valeur property. Par exemple, si `property == PROPERTY_START_TIME`, la méthode affectera `value[0]` au champ `m_startTime`. Sinon, appel de la méthode de la super-classe.

C.7.5 Class RealObject

La classe abstraite RealObject, sous-classe de RescueObject, n'a aucun champ, ni aucune méthode. Elle ne sert qu'à regrouper les classes qui vont définir les objets réels de la simulation.

C.7.6 Abstract Class MotionlessObject extends RealObject

Cette classe ne contient qu'une seule méthode. Elle représente les objets immobiles.

Méthode

- ⋆ `MotionlessObject location()`
  Renvoie l'objet lui-même.

C.7.7 Class DummyObject extends MotionlessObject

C'est la classe «DummyObject» ou, en français, la classe des faux objets, des objets factices. On peut éventuellement être amené à utiliser un tel objet (voir page 138) pour éviter de causer des erreurs de pointeur «null pointer exception».

Constructeur

⊙ **DummyObject()**
Initialise `m_id` à -2 et `m_time` à -1.

Méthodes

⋆ `int type(), int x(), int y()`
Renvoient respectivement `TYPE_DUMMY`, 0 et 0.

⋆ `MotionlessObject location()`
Renvoie l'objet lui-même.

⋆ `void input(int property, int[] value)`
Ne fait rien.

C.7.8 Abstract Class PointObject extends MotionlessObject

Cette classe définit les objets représentés par un point sur le graphe de la carte. Pour rappel, les routes en sont les arrêtes, alors que les carrefours, les bâtiments, les connexion aux rivières, ... en sont les noeuds. Note : les bâtiments sont des noeuds particuliers, qui sont définis au point suivant.

Champs

• `int m_x, int m_y`
Ces champs contiennent les positions x et y des objets de cette classe.

Méthodes

⋆ `int x(), int y()`
Renvoie la position x / y de l'objet.

⋆ `void setX(int value), void setY(int value)`
Assigne les champs `m_x` et `m_y` de l'objet.

⋆ `void input(int property, int[] value)`
Affecte `value[0]` au champ correspondant à la valeur property. Cette valeur peut, à ce niveau, être : `PROPERTY_X` ou `PROPERTY_Y`. Sinon, appel de la méthode de la super-classe.

C.7.9 Class Building extends PointObject

Cette classe représente les bâtiments de la simulation.

Champs

- `int m_fieryness`
  Représente l'intensité de l'incendie. Ce champ peut prendre comme valeurs : 0 - non incendié ; 1 - début d'incendie ; 2 - incendie moyen ; 3 - incendie important ; 4 - non utilisé ; 5 - incendie éteint, bâtiment légèrement consumé ; 6 - incendie éteint, bâtiment moyennement consumé ; 7 - bâtiment totalement consumé (considéré éteint).
- `int m_brokenness`
  Représente l'état du bâtiment. 0 : intact, 25 : partiellement endommagé, 50 : à moitié effondré, 100 : complètement effondré.
- `int[] m_entrances`
  Liste d'ID des objets (routes, noeuds) permettant d'entrer dans le bâtiment.
- `boolean m_entrancesAreChanged`
  Indique si `m_entrance` a changé par rapport à `m_entrancesSet`.
- `Domain m_entrancesSet`
  Ensemble d'entrées (Voir Domain au point C.8 page 145 pour plus de détails).
- `int m_buildingCode`
  Code de la structure du bâtiment. 0=bâtiment en bois, 1=bâtiment avec trame en acier, 2=bâtiment en béton armé.
- `int m_buildingAreaTotal`
  Taille de l'étendue du bâtiment.

Méthodes

- ⋆ `int` **type()**
  Renvoie le type de l'objet.
- ⋆ `int` **fieryness()**, `int` **brokenness()**,
  `int` **buildingCode()**, `int` **buildingAreaTotal()**
- ⋆ `Domain` **entrances()**
  Renvoie `m_entrancesSet`, mis à jour si `m_entrancesAreChanged` vaut `true`.
- ⋆ `void` **setFieryness(**`int value`**)**,
  `void` **setBrokenness(**`int value`**)**,
  `void` **setBuildingCode(**`int value`**)**,
  `void` **setBuildingAreaTotal(**`int value`**)**
  Méthodes classiques d'affectation.

⋆ void setEntrances(int[] value)
Cette méthode affecte d'abord true (false sinon) à m_entranceAreChanged si value est différent de m_entrances. Ensuite, si ce champ est à true, il affecte value à m_entrances.
⋆ void input(int property, int[] value)
String toString
Ces méthodes sont définies dans les classes précédentes.

C.7.10 Class FireStation, AmbulanceCenter, PoliceOffice et Refuge extends Building

Ces classes sont similaires quant à la méthode qu'elles définissent (elles n'ont pas de champs propres).

Méthode

⋆ int type()
Renvoie le type de l'objet.

C.7.11 Abstract Class Edge extends MotionlessObject

Cette classe définit une arrête du graphe représentant la carte.

Champs

- int m_head
ID du noeud initial de l'arrête.
- int m_tail
ID du noeud final de l'arrête.
- int m_length
Longueur de l'arrête.

Méthodes

⋆ RescueObject head(), RescueObject tail()
Renvoie l'objet dont l'ID est contenu dans m_head (m_tail). Si cet objet n'est pas défini, renvoie un DummyObject.
⋆ int length()
Renvoie le contenu du champ m_length().
⋆ void setHead(int value), void setTail(int value),
void setLength(int value)
Affecte la valeur value au champ correspondant.

⋆ void input(int property, int[] value)
Cette méthode est définie dans les classes précédentes.
⋆ int x(), int y()
Renvoie les coordonnées x et y du milieu de l'arrête.

C.7.12 Class Road extends Edge implements Cloneable

Cette classe représente les routes.

Champs

- int m_width
Largeur de la route en mm.
- int m_block
Largeur en mm des débris obstruant la route.
- int m_repairCost
Nombre de personnes requises pour déblayer la route.
- int m_linesToHead
Nombre de voies de circulation de la queue à la tête.
- int m_linesToTail
Nombre de voies de circulation de la tête à la queue.

Méthodes

⋆ int **type**()
Renvoie le type de l'objet.
⋆ int **width**(), int **block**(), int **repairCost**(),
int **linesToHead**(),int **linesToTail**()
Renvoie les valeurs des champs de même nom.
⋆ int **aliveLinesToHead**(), int **alivesLinesToTail**()
Renvoie le nombre de voies non obstruées dans les sens queue tête / tête queue.
⋆ int **blockedLines**()
Renvoie le nombre de voies de circulation bloquées.
⋆ double **linewidth**()
Renvoie la largeur d'une voie de circulation.
⋆ boolean **hasCars**()
Cette méthode est utilisée pour estimer quand des voitures sont sur la route, au lieu des propriétés *carsPassToHead* et *carsPassToTail* qui ne sont pas encore implémentées par le simulateur. Techniquement, cette méthode crée un prédicat qui évalue sur l'ensemble des objets humanoïdes (point C.7.18) s'il y en a au moins un (autre que lui-même)

sur cette route-ci.

* `void` **reduceBlock()**
  Décrémente la valeur `m_block` de `WIDTH_CLEARED_BY_CLEARING`.
* `Domain` **adjacentRoadSet()**
  Renvoie l'ensemble des routes adjacentes à celle-ci.
* `void` **addRoadsTo(`RescueObject node, Domain roadSet`)**
  Si `node` est un objet Node, alors, prendre toutes les routes dont `node` est une extrémité, et les ajouter à `roadSet`.
* `boolean` **isGap()**
  Cette méthode évalue si la route est un «trou», c'est-à-dire qu'elle n'a aucune voie de circulation disponible, alors que pour chacune de ses extrémités, au moins une route (adjacente) est utilisable (contient au moins une voie non obstruée). Cette méthode contient un bogue dans cette version (0.41) de YabAI : il faut remplacer dans la dernière ligne le «head» par «tail».
* `void` **setWidth(`int values`)**,
  `void` **setBlock(`int values`)**,
  `void` **setRepairCost(`int values`)**,
  `void` **setLinesToHead(`int values`)**,
  `void` **setLinesToTail(`int values`)**,
  `void` **input(`int property, int[]value`)**,
  `String` **toString()**
  Ces méthodes sont définies dans les classes précédentes.
* `Road` **copy()**
  Renvoie une copie de cet objet.

C.7.13 Class River extends Edge

Représente une rivière.

Méthode

* `int` **type()**
  Renvoie le type de l'objet.

C.7.14 Class Vertex extends PointObject

Cette classe représente les noeuds en tant que noeuds du graphe. Les noeuds vus en tant que support des objets de la simulation sont présentés au point suivant.

Champs

- `int[] m_edges`
  Ce champ contient la liste des ID des arrêtes qu'il connecte.
- `boolean m_edgesAreChanged`
  Ce champ indique si le champ `m_edge` a été modifié depuis la mise à jour de `m_edgeSet`.
- `Domain m_edgesSet`
  Ce champ contient l'ensemble des objets (arrêtes) connectés à lui même.

Méthodes

- ⋆ `Domain` **edges**`()`
  Renvoie l'ensemble des arrêtes connectées à l'objet lui-même. L'ensemble `m_edgeSet` est mis à jour s'il ne l'était pas.
- ⋆ `void` **setEdges**`(int[] value)`
  Si value est différent de `m_edges`, affecte `true` à `m_edgeAreChanged` et `value` à `m_edges`. Sinon, affecte `false` à `m_edgeAreChanged`.
- ⋆ `void` **input**`(int property, int[] value)`
  Cette méthode est définie dans les classes précédentes.

C.7.15 Class Node extends Vertex

Cette classe représente les noeuds, en tant que connexions d'objets aux arrêtes, tels qu'un bâtiment à une route, une rivière à une route, deux routes entres elles...

Méthodes

- ⋆ `int` **type**`()`
  Renvoie le type de l'objet.
- ⋆ `Road` **roadBetweenThisAnd**`(Node another)`
  Renvoie la route contenue entre ce noeud et le noeud `another` si elle existe. Sinon renvoie `null`.
- ⋆ `void` **input**`(int property, int[] value)`
  Cette méthode est définie dans les classes précédentes.

C.7.16 Class RiverNode extends Vertex

Représente les points particuliers des connexions des (aux) rivières.

Méthode

⋆ int type()
Renvoie le type de l'objet.

C.7.17 Abstract Class MovingObject extends RealObject

Cette classe représente les objets mobiles.

Champs

- int m_position
  Contient l'ID de l'objet sur (dans) lequel se trouve cet objet-ci.
- int m_positionExtra
  Contient la distance de cet objet au début (*to the head*) de la route. Il vaut 0 s'il est ailleurs que sur une route.
- int[] m_positionHistory
  Contient, dans l'ordre chronologique, la liste des ID sur (dans) lesquels cet objet est passé.
- int m_setTimePositionHistory
  Contient la valeur du temps lors de la dernière mise à jour de m_positionHistory.

Méthodes

⋆ int x(), int y()
Renvoie la position x / y de l'objet.

⋆ MotionlessObject **location**()
Renvoie l'objet inanimé (route, bâtiment...) sur lequel se trouve cet objet-ci. Le but est de pouvoir connaître l'endroit fixe au dessus duquel se trouve cet objet.

⋆ RescueObject **position**()
Renvoie l'objet dans (ou sur) lequel se trouve cet objet-ci (champ m_position). Un simple exemple permet de faire la distinction avec **location**() : supposons qu'un civil se trouve dans une ambulance se trouvant sur la route. Pour le civil, **position**() renverra l'ambulance, et **location**() renverra la route.

⋆ int **positionExtra**(), int[] **positionHistory**()
Renvoie les valeurs des champs de même nom.

⋆ void **setPosition**(int value),
void **setPositionExtra**(int value)
Affecte, avec la valeur value, les champs de même nom.

⋆ boolean **positionHistoryIsChangedNow()**
Renvoie (m_positionHistory==time()), c'est-à-dire renvoie true si l'historique des déplacements a été mis à jour à ce temps-ci.
⋆ void **setPositionHistory**(int[] value)
Si value est différent de m_positionHistory, cette méthode lui affecte sa valeur. De plus, elle affecte **time()** au champ m_setTimePosition-History.
⋆ void **input**(int property, int [] value)
Affecte value[0] au champ correspondant à la valeur property. Cette valeur peut, à ce niveau, être : PROPERTY_POSITION, PROPERTY_POSITION_EXTRA ou PROPERTY_POSITION_HISTORY. Sinon, appel de la méthode de la super-classe.

C.7.18 Abstract Class Humanoid extends MovingObject

Cette classe définit les objets de types humanoïdes.

Champs

- int m_stamina
Détermine le nombre d'actions que peut faire l'objet humanoïde (non utilisé dans le championnat de 2001).
- int m_hp
Les points de vie de l'objet humanoïde. 0 signifie mort.
- int m_damage
Représente l'intensité des dommages physiques que subit l'objet humanoïde.
- int m_buriedness
Représente le niveau d'ensevelissement de l'objet humanoïde.

Méthodes

⋆ int **stamina()**, int **hp()**, int **damage()**, int **buriedness()**
Renvoie la valeur des champs équivalents.
⋆ void **setStamina**(int value), void **setHp**(int value), void **setDamage**(int value), void **setBuriedness**(int value)
Affecte la valeur des champs équivalents.
⋆ void **input**(int property, int [] value)
Cette méthode est définie dans les classes précédentes.
⋆ String **toString()**
Renvoie la chaîne de caractères de la super-classe RescueObject + des

infos sur l'objet qui sont : (b :) son niveau d'ensevelissement (`m_buriedness`), (d :) les dommages qu'il subit (`m_damage`), (h :) ses points de vie restant (`m_hp`), (s :) sa vigueur, sa résistance (`m_stamina`), (pos :) l'ID de l'objet sur/dans lequel il se trouve.

C.7.19 Class Civilian, Car, AmbulanceTeam et PoliceForce extends Humanoid

Représente un civil, une voiture, une ambulance ou une voiture de police. Les classes Car, AmbulanceTeam, et PoliceForce sont similaires à celle-ci. La classe FireBrigade, légèrement différente, est traitée à part.

Méthode

- ⋆ `int type()`
  Renvoie le type de l'objet.

C.7.20 Class FireBrigade extends Humanoid

Représente une brigade de pompier.

Champs

- `int m_waterQuantity`
  Quantité d'eau disponible.
- `int m_stretchedLength`
  Longueur de tuyau déroulée.

méthodes

- ⋆ `int type()`
  Renvoie le type de l'objet.
- ⋆ `int waterQuantity()`, `int stretchedLength()`
  Renvoie la valeur des champs équivalents.
- ⋆ `void setWaterQuantity(int value)`,
  `void setStretchedLength(int value)`
  Affecte la valeur des champs équivalents.
- ⋆ `void input(int property, int[] value)`
  Affecte `value[0]` au champ correspondant à la valeur `property`. Cette valeur peut, à ce niveau, être : `PROPERTY_WATER_QUANTITY` ou `PROPERTY_STRETCHED_LENGTH`. Sinon, appel de la méthode de la super-classe.

C.8 Class Domain extends HashSet

Cette classe implémente un ensemble. Techniquement, c'est une extension de HashSet, qui est la classe qui implémente les ensembles à l'aide de tables de hachage. Cette classe Domain est principalement utilisée pour construire des ensembles d'objets RescueObject. Ces ensembles seront «la base de connaissance» des agents. Lorsqu'un agent voudra obtenir par exemple des informations sur un bâtiment, il ira dans son ensemble de bâtiments, y sélectionner l'objet bâtiment qui l'intéresse, pour l'interroger. Chaque agent aura bien sûr ses ensembles d'objets, puisque chacun aura des connaissances différentes.

C.8.1 Champ

- `final Domain EMPTY`
  Contient l'ensemble vide (initialisé par `new` **Domain**`(0,1)`)

C.8.2 Constructeurs

⊙ **Domain**`()`
Domain`(int initialCapacity)`
Domain`(int initialCapacity, float loadFactor)`
Domain`(Collection c)`
Ces trois premiers constructeurs sont basés sur le troisième, avec des valeurs par défaut (`loadFactor` vaut 0.75). `InitialCapacity` représente la capacité de l'ensemble, et `loadFactor` la charge de remplissage pour la table de hachage (si `initialCapacity` est trop élevé ou `loadFactor` trop bas, cela risque de dégrader les performances). Le troisième constructeur crée un nouvel ensemble vide ; l'instance de la table de hachage sur lequel il repose a la capacité et le taux de charge passés en paramètres. Le dernier construit un nouvel ensemble contenant les éléments de la collection `c`.

C.8.3 Méthodes

⋆ `Domain` **get**`(Predicate condition)`
Renvoie l'ensemble (peut-être vide) des objets vérifiant la condition.

⋆ `Object` **getMin**`(Function f)`, `Object` **getMax**`(Function f)`
Evalue tous les objets avec la fonction `f` et renvoie celui qui a donné la plus petite / grande valeur. Si l'ensemble est vide, renvoie `null`.

⋆ `void` **remove**`(Predicate condition)`
Enlève de l'ensemble tous les objets ne vérifiant pas la condition.

- ⋆ `boolean` **contains**`(predicate condition)`
  Renvoie `true` si l'ensemble possède un élément vérifiant la condition.
- ⋆ `int` **size**`(Predicate condition)`
  Renvoie le nombre d'objets vérifiant la condition.
- ⋆ `Object` **pop**`()`
  Renvoie un objet de l'ensemble et l'efface de celui-ci. Si l'ensemble est vide, renvoie `null`. Cette méthode est utilisée un peu comme `Iterator.next()`, afin d'obtenir tous les éléments de l'ensemble les uns après les autres (sauf qu'ici, ils sont enlevés de l'ensemble).
- ⋆ `Object` **popMin**`(Function f)`
  Renvoie l'objet minimisant la fonction `f` et l'enlève de l'ensemble.
- ⋆ `Domain` **locationSet**`()`
  Renvoie l'ensemble des *locations* des objets de cet ensemble-ci. Tous les objets de cet ensemble-ci doivent être des objets RescueObject.
- ⋆ `Domain` **union**`(Domain another)`
  Ajoute à un nouvel ensemble les objets de cet ensemble-ci, puis les objets de l'ensemble `another` qui n'y sont pas déjà présents. Renvoie ce nouvel ensemble.
- ⋆ `void` **add**`(Domain another)`
  Ajoute à cet ensemble-ci les objets de l'ensemble `another` qui ne sont pas déjà présents.
- ⋆ `Object` **getRand**`()`
  Renvoie un objet, de cet ensemble-ci, choisi au hasard.
- ⋆ `Object` **popRand**`()`
  Renvoie un objet, de cet ensemble-ci, choisi au hasard, et l'enlève de l'ensemble.
- ⋆ `String` **toString**`()`
  Revoie la chaîne de caractères commençant par la ligne «vvvvvvvvv», et finissant par «^^^^^^^^^». Une ligne est ajoutée entre ces deux-ci pour chaque objet de l'ensemble : elle commence par «> >», et est suivie du nom de l'objet (en général, *nom_de_classe*@*code_hexadécimal*).

C.9 Class ObjectPool

Cette classe est celle qui va contenir toute la base de connaissance des agents. En effet, elle est principalement faite de domaines (ensembles d'objets vu au point C.8) et de méthodes concernant leur organisation. Ainsi, chaque agent aura son ensemble de route, son ensemble de bâtiments, etc.

C.9.1 Champs

- `Hashtable m_pool`
  C'est la table qui range tous les objets en fonction de leurs ID. Pour obtenir un objet quelconque dont on connaît l'ID, c'est ici qu'il faut venir voir.
- `Domain m_roadSet`
  `Domain m_building`
  `Domain m_refugeSet`
  `Domain m_humanoidSet`
  `Domain m_civilianSet`
  `Domain m_fireBrigadeSet`
  `Domain m_ambulanceTeamSet`
  `Domain m_policeForceSet`
  Ce sont les ensembles des objets connus de notre agent, par catégories. Il est nécessaire d'utiliser ceci en plus de `m_pool` afin d'optimiser les recherches. Par exemple, une recherche sur les bâtiments en feu pourrait se faire via `m_pool`, mais il faudrait au préalable regarder tous les objets qui y sont pour voir s'ils ne sont pas des bâtiments... Les cinq derniers ensembles ne contiennent que les objets qui sont visibles.
- `Domain m_missedHumanoidSet`
  `Domain m_missedCivilianSet`
  `Domain m_missedFireBrigadeSet`
  `Domain m_missedAmbulanceTeamSet`
  `Domain m_missedPoliceForceSet`
  Comme les agents ont un champ de vision limité (10m), il y a des objets hors de leur vue. Lorsqu'un agent voit un objet mobile, il l'enregistre dans un des ensembles cités ci-dessus. Lorsque cet objet disparaît de son champ de vision, il est alors enregistré dans un de ces ensembles-ci, comme dernière information sensorielle perçue. Ces ensembles contiennent donc les objets mobiles tels qu'ils étaient la dernière fois qu'ils ont été vus.
- `Domain m_visibleBuildingSet`
  C'est l'ensemble des bâtiments vus par l'agent au moment présent.

- int m_numOfAmbulanceTeam
  int m_numOfFireBrigade
  int m_numOfPoliceForce
  C'est le nombre d'objets présents dans l'ensemble correspondant.
- RescueObject m_self
  C'est l'objet qui possède cette base de connaissance.
- int m_time
  C'est le temps auquel a eu lieu la dernière restructuration (voir point C.9.3).
- World m_world
  C'est le monde auquel appartiennent tous les objets.
- History m_history
  C'est l'historique. Voir la classe History au point C.18
- Controller m_owner
  C'est ce qui pilote (contrôle) l'agent (le centre de ses décisions). Voir chapitre C.19

C.9.2 Méthodes habituelles

⋆ int **numOfAmbulanceTeam**()
int **numOfFireBrigade**()
int **numOfPoliceForce**()
RescueObject **self**()
int **time**()
Domain **roadSet**()
Domain **buildingSet**()
Domain **humanoidSet**()
Domain **refugeSet**()
Domain **civilianSet**()
Domain **fireBrigadeSet**()
Domain **ambulanceTeamSet**()
Domain **policeForceSet**()
Domain **missedHumanoidSet**()
Domain **missedCivilianSet**()
Domain **missedFireBrigadeSet**()
Domain **missedAmbulanceTeamSet**()
Domain **missedPoliceForceSet**()
Domain **visibleBuildingSet**()
Toutes ces méthodes renvoient le contenu de leur champ respectif.

⋆ void **setOwner**(Controller owner)
Affecte au champ m_owner le contrôleur owner.

C.9.3 Autres méthodes

- ⋆ `void` **init()**
  Cette méthode garnit les 3 champs `numOf...` du nombre d'objets comptés dans les ensembles respectifs.
- ⋆ `RescueObject` **get(**`int id`**)**
  Renvoie l'objet dont l'ID est `id`.
- ⋆ `void` **put(**`RescueObject obj`**)**
  Ajoute l'objet `obj` dans la table `m_pool` (s'il existe déjà, l'ancien est écrasé).
- ⋆ `void` **restructure(**`int[] data, int selfId, int time`**)**
  Globalement, c'est la méthode qui met à jour la base de connaissance de l'objet `selfId` grâce au tableau `data[]` venu du noyau au temps time. Techniquement, `time` est affecté à `m_time`. Ensuite, l'ensemble `m_visibleBuildingSet` est vidé de ses objets. Le tableau `data[]` est analysé : à chaque fois qu'un type d'objet passe, on regarde s'il est dans la case de connaissances (`m_pool`). Sinon, on le crée, on lui initialise son ID, on le met dans `m_pool`, et si son ID est le même que `selfId`, on l'affecte à `m_self` (fin du sinon). On lui affecte le nouveau temps time, et on garnit ses champs de propriétés. Finalement, on l'ajoute dans un des ensembles s'il doit s'y trouver. Enfin, les humanoïdes présents dans l'ensemble `m_missedHumanoidSet` qui sont rentrés dans le champ de vision de l'agent sont enlevés de cet ensemble, et les méthodes **copeWithNarrowVisibleDistance()** et **lookAtOthers()** sont appelées.
- ⋆ `void` **copeWithNarrowVisibleDistance()**
  Il peut arriver que l'agent, dû à son champ de vision limité, ne voie même pas la route adjacente à celle où il est. S'il n'a pas encore vu cette route depuis le début de la catastrophe, alors il la considérera comme complètement obstruée. (Techniquement, si la prochaine route de son itinéraire n'a pas sa valeur `m_time` différente de celle du lancement de la simulation, alors il affecte à son champ `m_block` la valeur `m_width`.)
- ⋆ `void` **lookAtOthers()**
  Lorsqu'un agent voit un humanoïde arriver par une route qu'il considérait bloquée ou qu'il n'a pas encore explorée depuis la catastrophe, il peut directement en tirer comme conclusion qu'elle est praticable. Comme un agent peut analyser les objets mobiles qu'il voit, il est intéressant qu'il puisse profiter du chemin emprunté par ceux-ci pour connaître l'état des routes et mettre son ensemble de routes à jour. Techniquement, cette méthode lance **lookAtOthers(**`humanoid h`**)** sur chaque humanoïde de l'ensemble `humanoidSet`, et ce seulement si le temps est supérieur à 4 (avant, on estime que l'environnement change

trop).

⋆ `void` **lookAtOthers**`(Humanoid h)`
Cette méthode met à jour l'ensemble des routes de l'agent, en analysant le chemin emprunté par l'humanoïde `h` (qui est dans son champ de vision). Techniquement, cette méthode ne fait rien si `h` est resté en place, et/ou si la liste d'ID de son historique de position est inférieur à 2. Sinon, elle regarde toutes les routes de l'historique de position de `h`, et si ce sont des routes que l'agent (qui possède cet ObjectPool) n'a pas encore explorées ou qu'il pensait complètement obstruées, il les met à jour en considérant qu'une voie de circulation est libre. Attention, l'analyse de ce code dans la version actuelle (0.41) de YabAI recelle un bogue. Il faut rempacer cette partie du code

```
int block;
if (nd == rd.head()) {
   block = ((int) ((rd.linesToHead()-1) * rd.lineWidth())) - 1;
} else {
   block = ((int) ((rd.linesToTail()-1) * rd.lineWidth())) - 1;
}
```

par celle-ci

```
int block = (nd == rd.head())
? (int) (2 * rd.lineWidth() * (rd.linesToHead() - 0.5)) - 1
: (int) (2 * rd.lineWidth() * (rd.linesToTail() - 0.5)) - 1;
```

⋆ `RescueObject` **newRescueObject**`(int type)`
Crée un nouvel objet RescueObject de type type. Pour rappel, les différents types se trouvent au point C.6.1 page 125. Ils sont, par exemple, `TYPE_WORLD`, `TYPE_BUILDING`, `TYPE_CIVILIAN`... Notez que ces objets ne sont pas initialisés, outre l'affectation setMyWorld de l'objet (point C.7.3 page 134).

⋆ `void` **removeMissed**`(MovingObject obj)`
Enlève obj de `m_missedHumanoidSet`, et de `m_missedxxxSet` où `xxx` remplace `Civilian`, `FireBrigade`, `PoliceForce` ou `AmbulanceTeam`, selon le type de l'objet.

C.10 Interface CostFunction

C'est juste une interface qui définit la fonction d'évaluation d'un chemin.

⋆ `int` **eval**`(RescueObject from, RescueObject to)`
Donc cette fonction nous renvoie le coût du chemin entre l'objet `from` et l'objet `to`.

C.11 Class Route

Cette classe construit des chemins ou des listes chaînées d'objets. Pour bien comprendre comment un chemin est représenté, voyons sa structure de donnée. Un chemin est un ensemble de 4 champs : la tête du chemin (le point final), la queue du chemin, le coût d'utilisation de ce chemin, et sa longueur. Un chemin de base n'a alors pas de queue, sa longueur est de 1 et son coût de 0. Il n'est en fait qu'un point : l'endroit où on se trouve.

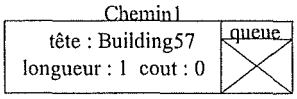


FIG. C.2 – Un chemin de base.

Ensuite, lorsqu'on va ajouter un deuxième objet à ce chemin (une nouvelle destination), on va reconstruire un objet route sur le premier, mettant le chemin de base dans la queue. La longueur sera incrémentée de 1, et le coût calculé (par défaut, il est aussi incrémenté de 1, mais cela peut varier en fonction de l'état de la route...).

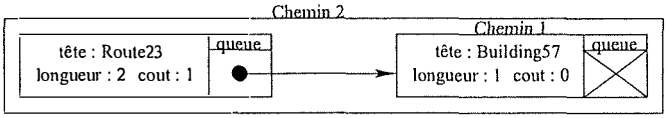


FIG. C.3 – Un chemin élémentaire.

Et ainsi de suite...

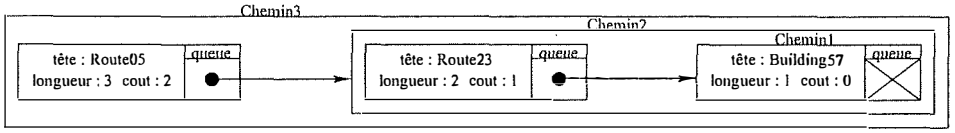


FIG. C.4 – Un chemin quelconque.

Notez bien que le chemin se lit à l'envers : l'endroit initial est Building57 et l'endroit final Route05 !

C.11.1 Champs

- `final Route EMPTY_ROUTE`
  C'est un chemin vide.
- `RescueObject m_obj`
  C'est objet de tête du chemin : son point final.
- `Route m_prev`
  C'est la queue du chemin : le chemin privé de son point final.
- `int m_sn`
  C'est le coût du chemin.
- `int m_length`
  C'est la longueur du chemin, ou encore le nombre d'objets qu'il comporte, ou encore le nombre de chemins qui le compose.

C.11.2 Constructeurs

⊙ **Route()**
C'est le chemin vide. Ce constructeur initialise les 4 champs de la manière suivante : `m_obj=null` ; `m_prev=null` ; `m_sn=0` ; `m_length=0`.

⊙ **Route(RescueObject obj)**
C'est le chemin de base. Ce constructeur initialise les 4 champs de la manière suivante : `m_obj=obj` ; `m_prev=null` ; `m_sn=0` ; `m_length=1`.

C.11.3 Méthodes Habituelles

⋆ `Route` **previous()**
Renvoie `m_prev`.

⋆ `boolean` **isEmpty()**
Renvoie (`m_prev==null`).

⋆ `int` **length()**
Renvoie `m_length`.

⋆ `int` **cost()**
Renvoie `m_sn`.

⋆ `String` **toString()**
Renvoie la chaîne de caractères suivante :
"len :" + `m_length`
(Pour chaque sous-chemin rt :) + "- - " + `rt.m_length` +" : " + `rt.m_obj` + "- - e :" + `m_obj`.

C.11.4 Autres Méthodes

- ⋆ `Route` **newExpandedRoute**`(RescueObject obj)`
  C'est cette méthode qui sert à agrandir la route. En fait, elle crée une nouvelle route, place en tête (`m_obj`) l'objet `obj`, incrémente la longueur (`m_length`) et le coût (`m_sn`) de 1, et place le chemin actuel dans la queue (`m_prev`) de la nouvelle route. Pour une explication graphique, voir au début de ce chapitre.
- ⋆ `Route` **newExpandedRoute**`(RescueObject obj, Domain dests, CostFunction cost)`
  Cette méthode est une variante de la précédente, en ce qui concerne le coût. Le coût du chemin créé aura en effet la valeur suivante : **costNextIs**`(obj, dests, cost)`, ou encore l'évaluation, par la fonction `cost`, de la distance entre l'avant-dernier point et le dernier.
- ⋆ `int` **costNextIs**`(RescueObject obj, Domain dests, CostFunction cost)`
  Renvoie le coût du chemin actuel plus le coût de la fin de ce chemin à l'objet `obj`. Techniquement, `m_sn` est augmenté (si `dests` contient `obj`) de l'évaluation, par la fonction `cost`, de la distance entre `m_obj` et `obj` (`cost.eval(m_obj,obj)`). Pour des détails sur CostFunction, voir chapitre C.10 page 150.
- ⋆ `RescueObject` **get**`(int n)`
  Renvoie le $n^{ième}$ objet dans le chemin (le dernier étant `m_obj`)
- ⋆ `RescueObject` **top**`()`
  Equivalent à **get**`(0)`.
- ⋆ `RescueObject` **end**`()`
  Renvoie le contenu de `m_obj`.
- ⋆ `RescueObject` **self**`()`
  Renvoie l'objet (un agent) qui a construit et donc possède ce chemin. Cette méthode renvoie donc le propriétaire de ce chemin-ci.
- ⋆ `Route` **routeFromStartingTo**`(int len)`
  Renvoie le sous-chemin de ce chemin-ci, dont la longueur est égale à `len`. Attention, un appel avec `len` ≤ 0 pourrait causer une erreur de pointeur nul !
- ⋆ `boolean` **contains**`(Predicate cond)`
  Renvoie `true` si le chemin contient un sous-chemin (pouvant être lui-même) vérifiant la condition `cond`.
- ⋆ `Domain` **expand**`(Predicate cond, Domain dests)`
  Renvoie l'ensemble de tous les objets directement accessibles à partir du point final de ce chemin, excepté l'objet étant à l'avant-dernier point de ce chemin. Dans le cas d'un bâtiment, ce sont ses entrées qui sont

renvoyées. Dans le cas d'une route, ce sont les deux routes adjacentes (au début et à la fin). Dans le cas d'un noeud, soit `cond=null`, et alors c'est l'ensemble des bâtiments sans incendies (`fieryness` est 0 ou plus de 4) et des routes (utilisables au moins dans un sens) qui est renvoyé ; soit `cond` $\neq$ `null`, et alors les objets à renvoyer sont évalués par `cond` (dans l'ensemble des arrêtes connectées au noeud).

- ⋆ `int[]` **toIdList()**
  Le but de cette méthode est d'obtenir un tableau d'ID utilisable par la classe IO pour communiquer le chemin que désire emprunter l'agent. Il est préférable de ne pas se servir de cette méthode autrement, car elle ne renvoie pas toujours exactement la liste des identifiants des objets contenus dans le chemin. En voici l'explication : si le noyau (et plus précisément, le simulateur de traffic) reçoit un chemin contenant une route qui est obstruée, l'agent qui désire l'emprunter ne pourra même pas atteindre le noeud auquel est connecté cette route. Or, les agents peuvent avoir besoin d'emprunter quand même ce noeud (par exemple un agent PoliceForce pour déblayer cette route). C'est pourquoi d'une part la méthode **copeWithBugOfTrafficSimulatorVer022_024()** (spécifiée un peu plus loin) a été créée, et d'autre part, cette méthode-ci ne renvoiet pas, pour les agents PoliceForce, l'ID du dernier objet de son chemin, si cet objet est une route. Donc les agents PoliceForce termineront toujours leurs déplacements sur des noeuds. Cela ne pose pas de problème puisqu'ils n'ont pas besoin d'avoir accès aux bâtiments ou d'être sur les routes elles-mêmes pour faire leur travail.
- ⋆ `Domain` **getRoadSet()**
  Renvoie l'ensemble de toutes les routes contenues dans ce chemin.
- ⋆ `Object[]` **copeWithBugOfTrafficSimulatorVer022_024()**
  Cette méthode renvoie deux objets : le premier est le plus long sous-chemin de ce chemin-ci (et de même origine) qui ne contient pas de route bloquée, et le deuxième est la première route bloquée qui est rencontrée en parcourant ce chemin (s'il n'y en a pas, il vaudra `null`).

C.12 Router

Cette classe est celle qui va nous permettre de nous repérer dans la carte du monde. En passant à ses méthodes, son point d'origine et les destinations possibles, celles-ci nous renvoient les chemins correspondants, cohérents et optimaux.

C.12.1 Final class RouteCostF extends Function

C'est une toute petite sous-classe de Function qui sera utilisée dans la classe Router. Elle ne fait qu'implémenter la fonction d'évaluation d'un chemin.

⋆ `int eval(Object obj)`
Cette fonction évalue donc le coût d'un chemin (`obj` doit être un chemin, un objet Route). Techniquement, elle renvoie `(Route)obj.cost()`.

C.12.2 Champ

• `final RouteCostF ROUTE_COST_F`
C'est la fonction d'évaluation du coût d'un chemin donné.

C.12.3 Méthodes

⋆ `Route get(RescueObject origin, Domain destinations, CostFunction cost)`
`Route get(RescueObject origin, Domain destinations, CostFunction cost, int n)`
`Route get(RescueObject origin, Domain destinations, CostFunction cost, Predicate cond)`
Ces trois méthodes ne sont qu'un appel vers la quatrième, avec `n=0` et `cond=null` s'ils ne sont pas spécifiés.

⋆ `Route get(RescueObject origin, Domain destinations, CostFunction cost, int n, Predicate cond)`
Cette méthode renvoie le chemin de coût minimum entre l'origine et l'ensemble des destinations. Par exemple, si la fonction de coût `CostFunction` est une fonction qui calcule la distance entre deux objets, la méthode renvera le chemin le plus court. Le predicat `cond` sert à éliminer des objets qu'on ne voudrait pas trouver sur le chemin (ex : un bâtiment en feu). Cependant, si la condition est trop forte (pas de chemin possible sous cette condition), elle sera ignorée. Si l'ensemble de destination est vide, la méthode renverra un chemin vide. L'entier `n` permet d'obtenir le $n^{ième}$ chemin de coût minimum. Par exemple, avec une fonction de coût de distance et avec 3 chemins, le premier d'une longueur de 30m, le deuxième de 35m et le troisième de 50m, un `n=0` renverra le chemin de 30m, un `n=1` celui de 35m, et un `n` égal ou supérieur à 2 renverra celui de 50m.

C.13 Un exemple de routage

Soit le graphe suivant (supposons que les agents ne se déplacent que sur les noeuds). Supposons un appel habituel de **get** : `get(`N_1`, {`N_6`, `N_7`}, Distance_vol_oiseau, 0, null)`. Notez que nous écrirons $[N_1 \rightarrow N_2]$ un chemin élémentaire d'origine N_1 et de fin N_2 pour plus de compréhension.

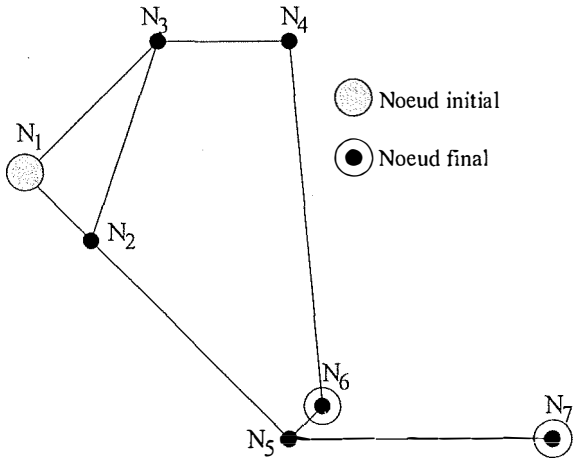


FIG. C.5 – Un simple graphe.

init $dest := \{B_1; B_2\}$
$open := \{\}$
$close := \{\}$
$open := \{[N_1]\}$

iter 1 Sélection du chemin de coût minimum (ici : le moins long) dans *open* : $[N_1]$
open est vide ? NON -→ ne rien faire.
$N_1 \subset dest$? NON -→ ne rien faire.
$open := \{\}$; $close := \{[N_1]\}$
$N_1 \subset dest$? NON -→ ne rien faire.
$expand([N_1]) = \{N_2, N_3\}$
N_2 coïncide-t-il avec la fin d'un chemin ? NON
-→ $open = \{[N_1 \rightarrow N_2]\}$
N_3 coïncide-t-il avec la fin d'un chemin ? NON
→ $open = \{[N_1 \rightarrow N_2], [N_1 \rightarrow N_3]\}$

iter 2 Sélection du chemin le moins long dans *open* : $[N_1 \rightarrow N_2]$
open est vide ? NON → ne rien faire.

$N_2 \subset dest$? NON $\rightarrow$ ne rien faire.
$open = \{[N_1 \rightarrow N_3]\}$; $close := \{[N_1], [N_1 \rightarrow N_2]\}$
$N_2 \subset dest$? NON $\rightarrow$ ne rien faire.
$expand([N_1 \rightarrow N_2]) = \{N_3, N_5\}$ (N_1 n'est pas dans le résultat, dû à la programmation interne de expand (voir point C.11.4, page 153).
N_3 coïncide avec la fin d'un chemin ? OUI (avec $[N_1 \rightarrow N_3]$ de *open*)
$\rightarrow$ Longueur de $[N_1 \rightarrow N_3]$ plus grande que celle de $[N_1 \rightarrow N_2 \rightarrow N_3]$?
NON $\rightarrow$ ne rien faire.
N_5 coïncide-t-il avec la fin d'un chemin ? NON
$\rightarrow$ $open = \{[N_1 \rightarrow N_3], [N_1 \rightarrow N_2 \rightarrow N_5]\}$

iter 3 Sélection du chemin le moins long dans *open* : $[N_1 \rightarrow N_3]$
open est vide ? NON $\rightarrow$ ne rien faire.
$N_3 \subset dest$? NON $\rightarrow$ ne rien faire.
$open = \{[N_1 \rightarrow N_2 \rightarrow N_5]\}$; $close := \{[N_1], [N_1 \rightarrow N_2], [N_1 \rightarrow N_3]\}$
$N_3 \subset dest$? NON $\rightarrow$ ne rien faire.
$expand([N_1 \rightarrow N_3]) = \{N_2, N_4\}$
N_2 coïncide avec la fin d'un chemin ? OUI (avec $[N_1 \rightarrow N_2]$ de *close*)
$\rightarrow$ La longueur de $[N_1 \rightarrow N_2]$ est-elle plus grande que celle de $[N_1 \rightarrow N_3 \rightarrow N_2]$?
NON $\rightarrow$ ne rien faire.
N_4 coïncide-t-il avec la fin d'un chemin ? NON
$\rightarrow$ $open = \{[N_1 \rightarrow N_2 \rightarrow N_5], [N_1 \rightarrow N_3 \rightarrow N_4]\}$

iter 4 Sélection du chemin le moins long dans *open* : $[N_1 \rightarrow N_3 \rightarrow N_4]$
open est vide ? NON $\rightarrow$ ne rien faire.
$N_4 \subset dest$? NON $\rightarrow$ ne rien faire.
$open = \{[N_1 \rightarrow N_2 \rightarrow N_5]\}$
$close := \{[N_1], [N_1 \rightarrow N_2], [N_1 \rightarrow N_3], [N_1 \rightarrow N_3 \rightarrow N_4]\}$
$N_4 \subset dest$? NON $\rightarrow$ ne rien faire.
$expand([N_1 \rightarrow N_3 \rightarrow N_4]) = \{N_6\}$
N_6 coïncide-t-il avec la fin d'un chemin ? NON
$\rightarrow$ $open = \{[N_1 \rightarrow N_2 \rightarrow N_5], [N_1 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6]\}$

iter 5 Sélection du chemin le moins long dans *open* : $[N_1 \rightarrow N_2 \rightarrow N_5]$
open est vide ? NON $\rightarrow$ ne rien faire.
$N_5 \subset dest$? NON $\rightarrow$ ne rien faire.
$open = \{[N_1 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6]\}$
$close := \{[N_1], [N_1 \rightarrow N_2], [N_1 \rightarrow N_3], [N_1 \rightarrow N_3 \rightarrow N_4], [N_1 \rightarrow N_2 \rightarrow N_5]\}$
$N_5 \subset dest$? NON $\rightarrow$ ne rien faire.
$expand([N_1 \rightarrow N_2 \rightarrow N_5]) = \{N_6, N_7\}$
N_6 coïncide avec la fin d'un chemin ? OUI (avec $[N_1 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6]$ de *open*)
$\rightarrow$ La longueur de $[N_1 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6]$ est-elle plus grande que celle de

$[N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_6]$? OUI
$\rightarrow$ *open* = $\{[N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_6]\}$
N_7 coïncide-t-il avec la fin d'un chemin ? NON
$\rightarrow$ *open* = $\{[N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_6], [N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_7]\}$

iter 6 Sélection du chemin le moins long dans *open* : $[N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_6]$
open est vide ? NON $\rightarrow$ ne rien faire.
$N_6 \subset$ *dest* ? OUI $\rightarrow$ *dest* vaut 1 ou n vaut 0 ? OUI
$\rightarrow$ Renvoyer le chemin $[N_1 \rightarrow N_2 \rightarrow N_5 \rightarrow N_6]$.

C.14 Class IO

C'est la classe qui s'occupe des connections et des communications avec le noyau.

C.14.1 Constructeur

⊙ **IO**(`InetAddress adress, int port`)
Construit un objet IO qui se connectera au port `adress` sur le port `port`.

C.14.2 Méthodes de configuration

Ces méthodes concernent la configuration des sockets.

⋆ `void` **doTimeout**`()`
Fixe le timeout (valeur d'attente maximale pour les communications) à la valeur par défaut (définie au point C.6.4 page 128). Techniquement, fait un appel à **setSoTimeout**(`IO_RECIEVE_OUT_TIME`).

⋆ `void` **doNotTimeout**`()`
Annule le timeout (permet une attente infinie). Techniquement, fait un appel à **setSoTimeout**(`0`).

⋆ `void` **setSoTimeout**(`int timeout`)
Fixe le timeout à la valeur `timeout`. 0 représente l'infini.

C.14.3 Méthodes de connexion

Ces méthodes sont celles utilisées dans les connexions au noyau et dans les analyses des informations envoyées par celui-ci. Elle sont principalement utilisées par le contrôleur (couche entre les agents et l'IO, voir C.19 page 165). Tout au long de ces méthodes, le booléen ignoreGIS sert uniquement dans le débogage pour ignorer les informations du GIS.

* void **clearTalkedMsgs**()
  Vide le contenu des messages échangés (contenus dans un champ privé).
* void **sendConnect**(int agentType, boolean ignoreGIS)
  C'est la méthode qui connecte un agent de type agentType au noyau. Le noyau signalera l'enregistrement de celui-ci par la méthode **receiveConnectOk**.
* ObjectPool **receiveConnectOk**(boolean ingoreGIS)
  Lorsque le noyau envoie son acceptation d'enregistrement de l'agent, l'ensemble des choses qu'il perçoit dans le monde l'accompagne. Ces informations sont traduites en groupes d'ensembles d'objets (ObjectPool, voir chapitre C.9 page 147), qu'il renvoie.
* void **sendAcknowledge**()
  Lorsque le noyau a envoyé son accord d'enregistrement avec tout ce qui est perçu dans le monde, il faut encore lui signaler qu'on l'a bien réceptionné, ainsi que les informations sur le monde. Cela se fait grâce à cette méthode.
* int[] **receive**()
  Cette méthode est équivalente à **receiveConnectOk** (que cette dernière utilise), sauf qu'elle ne crée pas d'ObjectPool, elle range juste les valeurs reçues (et pas forcément à la première connexion) dans un tableau d'entiers.
* int [] **udpsToLudp**(int [] [] udps)
  Méthode interne pour passer du protocole UDPS au protocole LUDP. Devrait être privée.
* int[] **loadData**(int[] data, boolean ignoreGIS)
  Méthode interne. Devrait être privée.

C.14.4 Méthodes d'actions

Ces méthodes sont celles qui sont supposées être utilisées par les agents.

* void **move**(Route route)
  Envoie au noyau l'ordre de déplacer l'agent le long de la route.
* void say(String message, Humanoid target)
  Envoie au noyau l'action de dire message à l'humanoïde target.
* void tell(String message)
  Envoie au noyau l'action de communiquer message par radio.
* void **extinguish**(Building target, int direction, int positionX, int positionY, int quantity)
  Envoie au noyau l'action d'éteindre le building target. La variable direction donne la direction vers laquelle est dirigée le jet d'eau (direction est en minutes, et 0 donne l'axe des Y). positionX et

positionY sont les positions de la bouche d'eau. quantity est la quantité d'eau qu'on désire déverser sur la cible.

* void **rescue**(Humanoid target)
  Envoie au noyau l'action d'embarquer l'humanoïde target (l'agent doit être une ambulance).
* void **unload**()
  Envoie au noyau l'action de décharger l'humanoïde que l'agent transportait (l'agent doit être une ambulance).
* void **clear**(Road target)
  Envoie au noyau l'action de nettoyer la route target (l'agent doit être une force de police).
* void **rest**()
  Envoie au noyau l'action de ne rien faire.

C.15 Final class Main

C'est la classe principale utilisée pour lancer les agents. Il n'est pas vraiment nécessaire de détailler cette classe. Tout ce qu'il en sera dit c'est que pour chaque agent qu'elle lance, elle crée un xxxController (un thread), avec xxx=Civilian, FireBrigade, FireStation... et qu'ensuite, elle lance Controller.go()

C.16 Class Message

Les messages échangés entre les agents dans YabAI sont définis dans cette classe. Il s'agit d'une chaîne de caractères composée d'au moins deux entiers. Le premier entier représente un prédicat, et les suivants, les objets (leurs ID) concernés par le prédicat. Par exemple, "4 1256" signifie que la route 1256 est nettoyée (car MSG_CLEAR_ROAD défini au point C.6.4 page 128 vaut 4).

C.16.1 Champs

- ObjectPool m_myWorld
  Contient le monde de l'émetteur. Il est nécessaire dans le sens où si on veut examiner les objets par les ID passés dans le message, cela doit se faire par rapport aux objets du monde de l'émetteur.
- Stringm_string
  Contient le message : des entiers séparés par un espace.
- int[] m_tokens
  Contient la décomposition en entiers du message. m_tokens[0] contient

le prédicat (le premier entier du message), et `m_tokens[1]`, l'ID d'un objet (le deuxième entier du message), et ainsi de suite.

- `RescueObject m_from`
  Contient l'objet émetteur du message.
- `int m_time`
  Contient le temps auquel a été émis le message.

C.16.2 Constructeur

⊙ **Message** `(ObjectPool myWorld, String message, RescueObject from)`
Construit un message. Le champ `m_myWorld` est rempli par `myWorld` ; `m_message` par `message`, et `m_from` par `from`. De plus, `m_tokens` sera garni de la décomposition en entiers de la chaîne de caractères `message`.

C.16.3 Méthodes

⋆ `int` **predicate()**
Renvoie le prédicat du message (`m_tokens[0]`, qui est le premier entier du string `m_string`).

⋆ `int` **tokenAt(**`int n`**)**
Renvoie la valeur `m_tokens[n]`.

⋆ `RescueObject` **getAt(**`int n`**)**
Renvoie l'objet (de `m_myWorld`), dont l'ID est la valeur trouvée dans `m_tokens[n]` (le n+1 ième entier du message).

⋆ `String` **get(**`int predicate, RescueObject obj`**)**
Si `obj` est `null`, renvoie `null`. Sinon, cette méthode renvoie un string d'entiers, séparés par un espace, dont le premier est `predicate`, et le suivant l'ID de l'objet `obj`, ou, s'il s'agit d'un civil, l'id de l'objet sur lequel il est (**obj.location().id()**).

⋆ `Domain` **get(**`int predicate, Domain set`**)**
Cette méthode renvoie un ensemble de messages. En effet, pour chaque objet de `set`, une chaîne de caractères est créée (avec la méthode précédente), et mise dans l'ensemble qui sera ensuite renvoyé.

⋆ `String` **toString()**
Renvoie le string `m_string`.

⋆ `String` **scopeString(**`String msg`**)**
Utilisé par le débogueur visuel (Scope).

C.17 Abstract class Action

Cette classe définit une action, c'est-à-dire un objet très simple comportant juste un champ `time`. Les sous-classes de cette classe spécialisent ces actions, telles que : se déplacer, déblayer une route, éteindre un incendie... Ces sous-classes ne sont guères plus compliquées. Elles ont juste un nom qui évoque le type d'action qu'elles représentent. Parfois, d'autres champs viennent se greffer, comme l'objet route qui a été déblayé, l'objet humanoïde qui a été secouru, etc...

Ces classes sont donc simplement de toutes petites bases d'informations sur ce qui s'est fait, quand cela s'est fait et où, voire avec qui. Et si on assemble toutes ces petites bases d'informations dans un ordre chronologique en fonction des actions exécutées par un agent, on obtient un historique de ses actions. C'est le rôle assumé par la classe History, chapitre C.18.

C.17.1 Abstract class Action

Champ

- `int m_time`
  C'est le temps auquel a été effectuée l'action.

Constructeur

⊙ **Action**(`int time`)
Construit une action dont le champ `m_time` vaut `time`.

Méthodes

⋆ `int` **time**()
Renvoie le temps auquel a été effectuée l'action.

⋆ `abstract String` **toScopeString**()
Utilisé par le débogueur visuel (Scope).

C.17.2 Class RestAction et UnloadAction

Ces classes étant similaires, elles sont présentées ensemble :

Constructeur

⊙ **RestAction**(`int time`) ; **UnloadAction**(`int time`)
Crée l'objet action correspondant et affecte `time` à `m_time`.

Méthodes

- ⋆ `String` **toString**()
  Renvoie le string " `AK_XXX` @"+time() avec `XXX = UNLOAD` ou `REST`.
- ⋆ `String` **toScopeString**()
  Utilisé par le débogueur visuel (Scope).

C.17.3 Class RescueAction et LoadAction

Champ

- `Humanoid m_target`
  Ce champ contient l'humanoïde qui est concerné par l'action.

Constructeur

- ⊙ **RescueAction**(`int time, Humanoid target`);
  LoadAction(`int time, Humanoid target`)
  Crée l'objet action correspondant et affecte `time` à `m_time`, ainsi que `target` à `m_target`.

Méthodes

Les mêmes que celles du point C.17.2.

C.17.4 Sous-classes restantes

Ces classes fonctionnent toutes comme les précédentes et ont exactement les mêmes méthodes. Aussi nous ne les présenterons que brièvement.

- La classe **ClearAction** a un champ `Road m_target`,
  et un constructeur **ClearAction**(`int time, Road target`).
- La classe **MoveAction** a un champ `Route m_target`,
  et un constructeur **MoveAction**(`int time, Route target`).
- La classe **TellAction** a un champ `String m_message`,
  et un constructeur **TellAction**(`int time, String message`).
- La classe **SayAction** a un champ `String m_message`, un champ `Humanoid m_target`,
  et un constructeur **SayAction**(`int time, String message, Humanoid target`).
- La classe **ExtinguishedAction** a les champs suivants :
  `Building m_target, int m_direction,`
  `int m_positionX, int m_positionY, int m_quantity`.

Pour plus d'info sur ces champs, voir la méthode extinguish de la classe IO, chapitre C.17.2, page 163. Son constructeur est le suivant : **ExtinguishAction**(int time, Building target, int direction, int positionX, int positionY, int quantity).

C.18 Class History

Cette classe s'occupe de garder l'historique des actions (par le biais de vecteurs).

C.18.1 Champs

- final int DEFAULT_SIZE
  C'est la taille d'un historique par défaut. Ce champ vaut 1.
- int m_maxSize
  Ce champ contient le nombre maximal d'actions qui seront mémorisées dans l'historique.
- Vector m_queue
  C'est la liste des actions.

C.18.2 Constructeurs

⊙ **History**()
Construit l'historique par défaut (Appel de **History**(DEFAULT_SIZE)).

⊙ **History**(int size)
Construit un historique de taille maximale size (size est affecté à m_maxSize, et un vecteur de taille size est créé et affecté à m_queue).

C.18.3 Méthodes

⋆ void **add**(Action act)
Ajoute l'action act à l'historique. Si celui-ci est plein, il efface la plus ancienne entrée avant d'y mettre l'action act.

⋆ boolean **isEmpty**()
Dit si l'historique est vide (**m_queue.isEmpty**()).

⋆ int **size**()
Renvoie la taille (nombre d'actions) de l'historique (**m_queue.size**()).

⋆ Action **refer**(int nth)
Renvoie la $nth^{ième}$ dernière action enregistrée, si elle existe. Sinon renvoie null. Par exemple si l'action A est enregistrée en premier, puis la

B, la C et enfin la D, refer(3) renverra B, alors que refer(5) renverra null.

⋆ `Action` **last()**
Renvoie la dernière action effectuée (**refer(1)**).

⋆ `String` **toString()**
Renvoie le string suivant :
"vvvvvvvvv\n"
[Pour chaque action a de `m_queue`] + "> > @"+ **a.time()**+ " " + a + "\n" + "^^^^^^^^^^".

C.19 Abstract class Controller

C'est la classe (abstraite) qui permet de manipuler (contrôler) les agents au sein du monde simulé. Cette classe est la couche située entre la couche IO qui discute avec le noyau, et la couche de l'agent lui-même (de sa «matière grise»). C'est donc l'avant-dernière étape, avant d'étudier le système multiagent lui-même. C'est un peu l'interface sur laquelle les agents vont venir se greffer (notez bien qu'il n'y a aucune prise de décision à ce niveau-ci !). Au point C.20, on verra une implémentation toute simple d'un agent : un civil qui ne fait que courrir vers un refuge.

C.19.1 Champs

Les champs de base dont doit disposer tout agent sont : son objet de communication avec le noyau (`m_io`), sa représentation du monde pour s'y orienter et y prendre des décisions raisonnables (`m_myWorld`), une représentation de lui-même (`m_self`), pour ne pas, par exemple, avoir à chercher après lui-même pour savoir où il se trouve, et enfin une carte pour lui permettre de se repérer dans les rues (`m_router`). A cela viennent s'en greffer d'autres, notamment pour le cas où un seul contrôleur pilote plus d'un agent.

- `IO m_io`
  C'est l'objet de communication avec le noyau.
- `ObjectPool m_myWorl`
  C'est la représentation du monde de l'agent.
- `RescueObject m_self`
  C'est la représentation que l'agent a de lui-même.
- `Router m_router`
  C'est la carte qui va lui permettre de trouver son chemin.
- `DistanceF m_distanceFromSelf_F`
  C'est une fonction qui, évaluée avec un objet, donne la distance entre

l'agent lui-même et cet objet.

- `Vector m_ctrlr`
  Ce champ contient tous les contrôleurs que pilote ce contrôleur-ci. Il y a donc bien sûr lui-même, mais il peut aussi piloter d'autres agents.
- `Integer m_actCounter`
  Utilisé pour le débogage.
- `Random m_rnd`
  C'est un générateur de nombres aléatoires.
- `Scope m_scope`
  C'est une fenêtre externe pour suivre les décisions des agents en cas de débogage.
- `int m_numBuried`
  `int m_numDeaed`
  Utilisé pour le débogage.

C.19.2 Méthodes habituelles

Comme dans nombreuses autres classes examinées auparavant, ces méthodes sont celles qu'on retrouve à chaque fois pour rendre compte de l'état des variables du contrôleur.

⋆ `ObjectPool` **world()**
Renvoie la représentation du monde qu'en a l'agent (`m_myWorld`).

⋆ `int` **time()**
Renvoie le temps qu'a l'agent (**m_myWorld.time()**).

⋆ `History` **history()**
Renvoie l'historique des actions de l'agent (`m_history`).

C.19.3 Constructeur

⊙ **Controller**(`int agentType, InetAddress address, int port, boolean ignoreGIS`)
Construit un contrôleur d'agent. Ainsi les variables `m_rnd` et `m_router` sont initialisées ; un objet IO est créé grâce aux paramètres `address` et `port`, et va connecter un agent de type `agentType` au noyau, qui va permettre de garnir `m_myWorld` et `m_self` ; le contrôleur ainsi créé est ajouté à la liste des contrôleurs `m_ctrlr` ; `m_distFromSelf_F` est initialisé avec **DistanceF**(`m_self`) ; et éventuellement, le scope est affiché (en fonction de l'état de la constante `USING_SCOPE`).

C.19.4 Méthodes principales

* `void` **go()**
  Voici donc la commande qui est lancée par la classe Main lorsqu'un contrôleur est créé pour un ou plusieurs agents. En fait, cette méthode démarre un thread (voir **run()**), pour chaque contrôleur dans la liste de contrôleurs (`m_ctrlr`).
* `abstract void` **prepareToAct()**
  Cette méthode est appelée dans **run()** dès que le noyau envoie de nouvelles informations concernant le monde. Cette méthode ne sera appelée qu'une seule fois, au tout début de la simulation. Elle permet de placer du code à exécuter (analyses préliminaires, par exemple) avant le début de la simulation.
* `abstract void` **act()**
  Cette méthode est à surcharger pour faire prendre des décisions à l'agent. C'est ici que se trouvera le siège de sa réflexion, sa planification...
* `abstract void` **communicate(Message message)**
  Cette méthode devra implémenter l'action à exécuter si l'agent reçoit un message.
* `void` **run()**
  C'est la méthode qui décrit le fonctionnement de base des agents. La voici en détail. Elle commence par créer deux champs :
  - `boolean actable = false.` qui spécifie si l'agent peut agir (au début il ne le peut pas, car il n'a pas encore reçu les infos du noyau).
  - `int numHearingATime = 0.` c'est le nombre de messages que l'agent entend à la fois (il ne peut en traiter que 4).
  Ensuite, c'est donc une boucle infinie qui fait les choses suivantes (en résumé) :
  1. récupération des informations venant du noyau.
  2. Si ces informations sont sensorielles (au début de chaque tour) :
     (a) Restructuration de la base de connaissance de l'agent (voir page 149).
     (b) Lancement de la méthode **prepareToAct()**.
     (c) `actable` reçoit l'affectation vrai.
     (d) Si `time==3`, une `InterruptedIOException` est levée (pt. 4).
     (e) Le timeout est lancé. Si les agents ne décident pas assez vite, une `InterruptedIOException` est levée (pt. 4).

3. Si ces informations sont auditives (que l'agent entend par un autre) :
   (a) Identification de l'objet émetteur.
   (b) Si c'est lui-même, un civil ou que c'est plus du $4^{ième}$ message qu'il entend, ne rien faire.
   (c) Sinon, récupérer le message et lancer la méthode **communicate(message)**.
4. Si une exception `InterruptedIOException` est levée et que l'agent peut agir (`actable == true`)
   (a) Fixer `numHearingATime` à 0.
   (b) Essayer, sous peine de voir une SendingCommandException levée :
   Si l'agent est un humanoïde (pas un bâtiment) et que sa vie est à zéro, appel de **rest()**.
   Lancement de la méthode **act()**.
   Lancement de la méthode **rest()**.
   (c) Lancement de la méthode **realTalk()** (voir page 170).
   (d) `actable` reçoit l'affectation `false`.
   (e) Le timeout est désactivé.
   (f) Eventuellement mise à jour du scope.

C.19.5 Deux simples sous-classes

Class SendingCommandExcception extends Exception

Une simple définition d'une exception, rien de particulier, excepté une méthode, **retAct()** qui lève une exception.

Class PriorityF extends Function

Cette classe implémente juste la fonction d'évaluation qui renvoie l'ID de l'objet qui lui est passé

⋆ `int` **eval(Object obj)**
Renvoie **obj.id()**.

C.19.6 Champs supplémentaires

- `final PriorityF PRIORITY_F`
  Contient la fonction d'évaluation de l'ID d'un objet.

- `Domain m_initCivBuilSet`
  A l'initialisation, ce champ est garni de tous les civils présents dans des bâtiments. Ce champ contient donc l'ensemble des civils présents dans des bâtiments à l'initialisation, et qui y sont peut-être encore ensevelis. De cet ensemble sont enlevés les civils dont on sait qu'ils ne sont pas sous des décombres.
- `Domain m_talkingMsgSet`
  Ce champ contient l'ensemble des messages que va envoyer par radio l'agent.
- `HashMap m_reportedRoadTimeMap`
  C'est une table de hachage un peu spéciale. D'habitude (dans ce programme), on donne un entier et on reçoit un objet. Ici, c'est l'inverse : on donne un objet Road et on reçoit un entier. Cet entier représente la valeur du temps auquel la route a été mise dans la table de hachage (et donc auquel un appel pour déblayer une route a été fait).
- `final Function m_reportedTimeIntervalF`
  Ce champ permet l'évaluation d'objets Road par la méthode **reportedTimeInterval** (voir point C.19.7). Elle prend la valeur de la route dans `m_reportedRoadTimeMap` et lui enlève le temps actuel. Cette fonction est boguée, elle devrait faire l'inverse.
- `Domain m_checkedEasyInj`
  Cet ensemble contient l'ensemble des civils blessés non enterrés.

C.19.7 Méthodes utilitaires

Notez qu'aucune de ces méthodes n'est obligatoirement utilisée. Hormis **realTalk** qui est utilisé dans **run()**, les autres sont appelées par les agents spécialisés (tous les xxxController avec xxx=Civilian, PoliceForce, ...). Il n'y a pas de prise de décision à ce niveau sans appel par un agent spécialisé. En ce qui concerne **realTalk**, cette méthode envoie des messages contenus dans `m_talkingMsgSet`. Ce champ n'est garni que par les appels des agents spécialisés. Donc si les agents spécialisés ne font rien, cette méthode ne fera rien non plus.

- ⋆ `void` **memorizeInitCivBuil()**
  Cette méthode garnit le champ `m_initCivBuilSet` par l'ensemble des civils (ceux en vue de **world()**`.m_civilianSet` et les autres de **world()**`.m_missedCivilianSet`) qui sont dans des bâtiments.
- ⋆ `void` **checkNoCivilian()**
  Si **time()** est strictement inférieur à `TIME_PREPARING_SIMULATION`, ne fait rien. Sinon, cette méthode examine tous les bâtiments visibles par l'agent. Pour chaque bâtiment, s'il est censé contenir des civils (analyse

du champ `m_initCivBuilSet`, qu'il vaut mieux avoir initialisé avant), et qu'ils n'y sont plus, ou qu'ils ne sont pas blessés (`damage=0`), alors les enlever du champ `m_initCivBuilSet` et envoyer un message radio (**Tell**) pour dire qu'il n'y a pas de civil blessé (`MSG_THERE_IS_NO_INJ`).

* `void` **reportInj**`(Domain inj)`
  Cette méthode sert à envoyer des messages radio (**Tell**) sur les blessés contenus dans `inj`. Elle commence par parler des blessés graves (nombre de tours avant de mourir<18), puis, si son quota de messages n'est pas dépassé, des blessés moyens (nombre de tours<330-**time**`()`), puis, si son quota de messages le permet, des blessés légers (les autres).
* `void` **realTalk**`()`
  Cette méthode prend tous les messages contenus dans `m_talkingMsgSet` et les envoie par radio (**Tell**) dans l'ordre suivant : les demandes de nettoyage de route, les blessés légers, les routes nettoyées, les blessés graves, les blessés normaux, `MSG_REPORT_CLEARED`, et si **time**`()` >= `TIME_SAVE_POWER_MSG`, les blessés soignés et les non blessés. Il est à noter aussi que si l'agent est un pompier, la méthode **repeatReportImportantRoad**`()` est lancée entre les messages du deuxième type et les messages du troisième.
* `int` **reportedTimeInterval**`(Road road)`
  Cette méthode renvoie la valeur contenue dans l'entrée `road` de la table de hachage `m_reportedRoadTimeMap` moins le temps actuel. En d'autres termes, elle renvoie le temps auquel la demande de déblayage de la route `road` a été faite, moins la valeur du temps actuel. C'est donc une valeur négative, ce qui est un bogue !
* `void` **repeatReportImportantRoad**`()`
  Pour toutes les routes qui sont une clé de la table de hachage `m_reportedRoadTimeMap` (en d'autres mots : pour toutes les routes qui ont reçu une demande de déblayage), enlever celles qui sont toujours bloquées ou qui ont la valeur `reportedTimeInterval` >= `TIME_FORGETTING_UNMOVABLE` (ce qui n'arrive jamais. Cette méthode est boguée !). Ensuite, pour celles qui restent, si le message de déblayage est enlevé de `m_talkingMsgSet`, refaire la demande (**Tell**).
* `void` **report EasyInj**`()`
  Cette méthode comptabilise tous les civils qui ne survivront pas d'ici la fin de la simulation (300 tours - nbr de tours déjà écoulés) mais qui ne sont pas ensevelis, et en informe tout le monde (**Tell**).

C.19.8 Méthodes d'actions

- ⋆ `void say(String message, Humanoid target)`
  Cette méthode soulève une exception pour signaler au thread la fin du tour.
- ⋆ `void` **tell**`(String message)`
  Ajoute le message `message` à l'ensemble des messages (`m_talkingMsgSet`) qui seront envoyés par radio. Ce n'est pas un appel direct au noyau, car les messages doivent passer par la politique de gestion de leur importance (Voir **realTalk**`()` point C.19.7).
- ⋆ `void` **tell**`(Set messages)`
  Idem que la précédente, appliquée à chaque message de l'ensemble `messages`.
- ⋆ `void` **move**`(Domain destinations)`
  Si `destinations` n'est pas vide, appel de la méthode **move** ci-dessous avec le chemin renvoyé par **routeTo**`(destinations)` (méthode définie au point C.19.9).
- ⋆ `void` **move**`(Route route)`
  Cette méthode commence par arranger le chemin pour qu'il ne crée pas de bogue (**copeTrafBugWithShortRoute**). Ensuite, elle envoie sa demande de déplacement à l'objet `m_io`, qui va s'occuper de la transmettre au noyau. Finalement, l'exception est levée (**retAct**`()`) pour signifier la fin du tour au thread.
- ⋆ `Route` **copeTrafBugWithShortRoute**`(Route route)`
  Si le chemin route comporte exactement 2 éléments, tel que l'origine de ce chemin soit une route et la destination un noeud, alors il faut faire quelques modifications au chemin, à cause d'un bogue du simulateur de trafic. Ce qu'on fait, c'est qu'on ajoute à la fin du chemin une route x, non obstruée et adjacente à l'origine, et puis on rajoute encore à la fin de ce nouveau chemin le noeud final du chemin original.
- ⋆ `void` **rest**`()`
  Envoie la demande rest à l'objet `m_io`, qui va s'occuper de la faire parvenir au noyau. Ensuite l'exception est levée.

C.19.9 Outils concernant les chemins et l'orientation

Fonctions

- ⋆ `CostFunction DISTANCE_CF`
  `int` **eval**`(RescueObject from, RescueObject to)`
  C'est la fonction utilisée pour l'évaluation de la distance entre ces 2 RescueObject. Si la destination finale (`to`) n'est pas une route, l'évaluation

donne 1. Sinon elle renvoie la distance (méthode **distance**(`from,to`) du point C.19.9) multipliée par :
- 100 s'il n'y a pas de chemin dégagé qui joint les deux objets ;
- 10 si l'agent n'a pas assez d'information sur l'état du chemin ;
- 1 s'il y a un chemin dégagé joignant les deux objets.

⋆ `CostFunction m_randDistance_CF`
`int` **eval**(`RescueObject from, RescueObject to`)
Cette fonction renvoie une évaluation de distance quelconque, une valeur entre 0 et distance(from,to).

Méthodes

Ce chapitre nous offre un jeu de méthodes pouvant être utilisées dans diverses conditions. Ainsi, la fonction de base sera **routeTo**(`Domain`), qui permet d'obtenir un chemin exempté de bâtiments en feu et dont la première route, si elle est bloquée, lancera une demande radio pour la débloquer. Si on ne désire pas la condition d'élimination des bâtiments en feu, on utilisera alors plutôt **dangerousRouteTo**(`Domain`). Il y a aussi moyen de donner une autre fonction de coût que celle de base, cette dernière qui calcule la distance à vol d'oiseau et multiplie le résultat par un facteur en fonction de l'encombrement du chemin.

⋆ `Route` **routeTo**(`Domain destinations`)
Renvoie un chemin jusqu'à `destinations`, exempté de bâtiments en feu, avec la fonction de coût de base. Demande un déblayage (**Tell**) si la première route du chemin est bloquée. Techniquement, renvoie routeTo(destintions, `DISTANCE_CF`) (Voir point C.19.9 pour `DISTANCE_CF`).

⋆ `Route` **routeAddingRandomeTo**(`Domain destinations`)
Renvoie un chemin jusqu'à `destinations`, exempté de bâtiments en feu, avec une fonction de coût aléatoire. Demande un déblayage (**Tell**) si la première route du chemin est bloquée. Techniquement, renvoie **routeTo**(`destination, m_randDistance_CF`).

⋆ `Route` **routeTo**(`Domain destinations, CostFunction cost`)
Renvoie un chemin jusqu'à `destinations`, exempté de bâtiments en feu, dont la fonction de coût est `cost`. Demande un déblayage (**Tell**) si la première route du chemin est bloquée. Techniquement, renvoie **checkSafetyRoute**(`dangerousRouteTo(destination,cost)`)).

⋆ `Route` **dangerousRouteTo**(`Domain destination`)
Renvoie un chemin jusqu'à `destinations`, avec la fonction de coût de base. Demande un déblayage (**Tell**) si la première route du chemin est bloquée. Techniquement, renvoie **dangerousRouteTo**(`destination, DISTANCE_CF`) (Voir point C.19.9 pour `DISTANCE_CF`).

- ⋆ `Route` **dangerousRouteTo**`(Domain destinations, CostFunction cost)`
  Renvoie le chemin entre la position de l'agent et `destinations`, avec la fonction de coût `cost`. Demande un déblayage (**Tell**) si la première route du chemin est bloquée. Techniquement, cette méthode crée un chemin via le router (**m_router.get(m_self.location()**, `destinations, cost`)), voir C.12.3), le vérifie pour éviter les bogues (**copeWithBugOfTrafficSimulatorVer022_024()**, voir C.11.4), puis, si le chemin mène directement sur une route bloquée, il y a demande de déblayage (**Tell**), et si c'est un pompier, ajoute le temps auquel a eu la demande dans `m_reportedRoadTimeMap` (voir C.19.6). Enfin, renvoie la route.
- ⋆ `Route` **checkSafetyRoute**`(Route route)`
  Vérifie si le chemin `route` ne comporte pas de bâtiments en feu. Si tel est le cas, toute la partie du chemin du premier building en feu à la fin, est enlevée du chemin. Le chemin ainsi révisé est renvoyé.
- ⋆ `final int` **distance** `(RescueObject from, RescueObject to)`
  Cette méthode renvoie la distance à vol d'oiseau entre les coordonnées des deux objets ($distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).
- ⋆ `final int` **direction**`(RescueObject origin, RescueObject target)`
  Cette fonction renvoie, en minutes, la valeur de l'angle entre la droite des Y et la droite entre `origin` et `rescue`.

C.19.10 Final class DistanceF extends Function

Champ

- `RescueObject m_org`
  Contient l'objet à l'origine de la distance à calculer.

Constructeur

- ⊙ **DistanceF**`(RescueObject org)`
  Crée une fonction qui permet d'évaluer la distance entre l'objet qui lui sera passé et l'objet `org`. Techniquement, affecte `org` à `m_org`.

Méthodes

- ⋆ `int` **eval**`(Object obj)`
  Renvoie la distance à vol d'oiseau entre `obj` et `m_org`. Techniquement, renvoie **distance**`(m_org, obj)`.
- ⋆ `void` **setOrg**`(RescueObject org)`
  Fixe l'origine. Techniquement, affecte `org` à `m_org`.

C.19.11 Méthodes de débogage

Ce sont des fonctions d'impression à l'écran, sauf putInterval qui est une fonction d'attente. Ces fonctions ne sont pas détaillées.

- ⋆ void **printState**()
- ⋆ void **printShortState**()
- ⋆ void **putInterval**(long millis)
- ⋆ void **prln**(String str)
- ⋆ void **pr**(String str)
- ⋆ void **prln2**(String str)
- ⋆ void **pi**(String str)
- ⋆ void **dp**(String str)

C.20 Class CivilianController

Voici donc la classe d'un agent très simple, qui ne fait que se diriger vers un refuge.

C.20.1 Champ

- int numOfAgenst
  Sert de compteur pour connaître le numéro du dernier agent lancé.

C.20.2 Constructeur

- ⊙ **CivilianController**(InetAddress address, int port, boolean ignoreGIS)
  Appelle le constructeur de la super classe, avec comme premier paramètre AGENT_TYPE_CIVILIAN.

C.20.3 Méthodes

- ⋆ Civilian **self**()
  Renvoie l'objet qui représente l'agent lui-même. Techniquement, renvoie m_self (de la super classe Controller).
- ⋆ void **prepareToAct**()
  void **communicate**()
  void **tell**(String message)
  Ces méthodes sont vides et ne font rien.
- ⋆ void **act**()
  Globalement, cette méthode fait se déplacer le civil vers un refuge.

Techniquement, fait un appel à `Controller.move(world().refuge-Set())`.

C.20.4 Analyse d'un mouvement

Il peut être intéressant de voir toute la série d'appels qui sont effectués lorsqu'on fait un simple appel à la méthode **move**. La méthode appliquée est celle soulignée, et se retrouve à [chapitre, page]. Analysons la demande de déplacement du civil vers son refuge : `move(world().refugeSet())`

```
move(world().refugeSet())                                          [C.19.2, 166]
move(m_myWorld.refugeSet())                                        [C.9.2, 148]
move(ens_rfgs)                                                     [C.19.8, 171]
move(routeTo(ens_rfgs))                                            [C.19.9, 172]
move(routeTo(ens_rfgs, DISTANCE_CF))                               [C.19.9, 172]
move(checkSafetyRoute(
       dangerousRouteTo(ens_rfgs, DISTANCE_CF)))                   [C.19.9, 172]
move(checkSafetyRoute(m_router.get(
       m_self.location(), ens_rfgs, DISTANCE_CF)))                 [C.7.17, 142]
move(checkSafetyRoute(
        m_router.get(pos, ens_rfgs,DISTANCE_CF)))                  [C.12.3, 155]
move(checkSafetyRoute(chemin_rfgs))                                [C.19.9,173]
move(chemin_safe_rfgs)                                             [C.19.8,171]
m_io.move(
        copeTrafBugWithShortRoute(chemin_safe_rfgs))               [C.19.8,171]
m_io.move(chemin_safe_nonbug_rfgs)                                 [C.14.4,159]
```

C.21 Class Scope

Ce chapitre ne sera pas, comme les précédents, un chapitre qui analysera le contenu de cette classe, mais plutôt un chapitre pour expliquer ce qu'est le *scope* et comment s'en servir.

C.21.1 Description

Il est impossible de développer des agents sans obtenir différentes informations au sujet de ces agents telle que son modèle interne du monde, l'action qu'il effectue, le chemin qu'il désire emprunter, les messages qu'il reçoit des autres agents... Pour toutes ces raisons, un débogueur visuel a été développé,

car l'information visuelle est plus facile à manipuler que l'information textuelle. C'est cet outil que la classe Scope implémente.

C.21.2 Lancement du scope

Pour pouvoir utiliser le *scope*, il y a quelques manipulations à faire dans le code. Il faut en fait changer deux valeurs de la classe Constants. Tout d'abord, la ligne

```
static final USING_SCOPE = false
```

doit être remplacée par

```
static final USING_SCOPE = true,
```

pour signaler l'activation du scope. Mais cela ne suffit pas. Il faut aussi dire quel agent on désire surveiller (sinon, on risque d'obtenir des erreurs de pointeur nul). Pour cela, il faut connaître l'ID de l'agent à surveiller. Les ID des agents sont affichés lors de leur création au lancement de YabAI. Une autre manière d'obtenir leur valeur est de regarder sur le «kuwataviewer» fourni avec le simulateur 0.4. Dès que l'id est connu, il faut remplacer, dans la ligne

```
static final int SCOPING_ID = 362 ;
```

le nombre de droite ń362ż par l'ID de l'agent à surveiller. De plus, la fonction `act()` devra contenir les lignes suivantes (qui auraient pu être ajoutées dans Controller) :

```
if (USING_SCOPE && self().id() == SCOPING_ID)
m_scope.reset(SCOPE_FIRE_TARGET) ;
```

C.21.3 Interpréter l'information visuelle

Le scope est une petite fenêtre, comme suit :

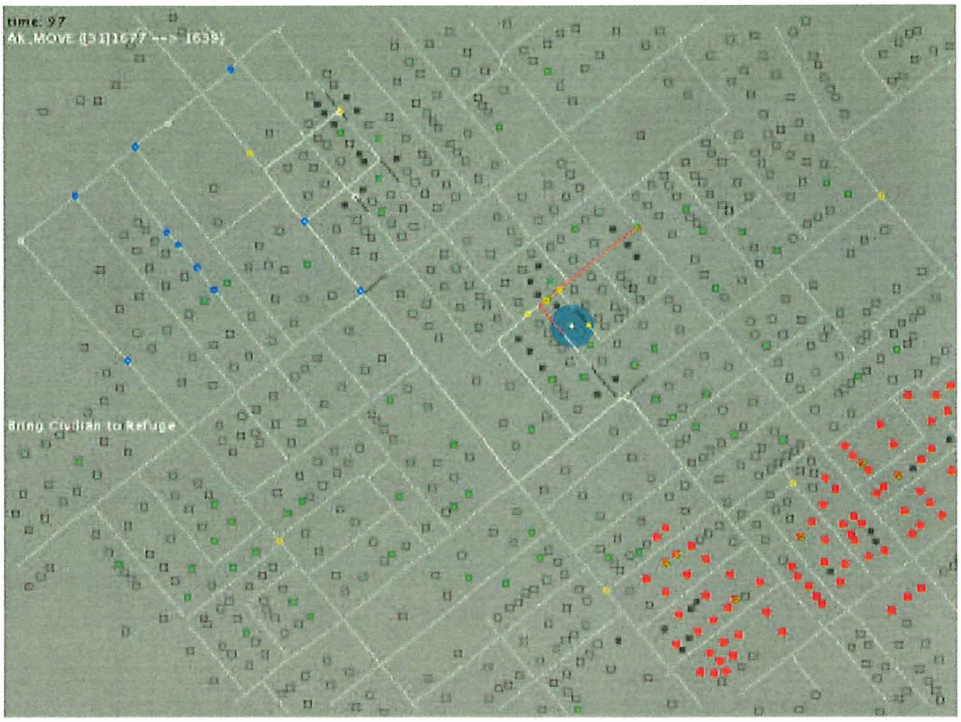


FIG. C.6 – Débogueur visuel de YabAI.

Dans cette fenêtre, les droites représentent les routes et les carrés représentent les bâtiments. Les petits cercles verts sont des civils, les bleus des pompiers, les blancs des ambulances et les jaunes des policiers. Le point dans le cercle est l'agent surveillé. S'il s'agit d'un agent pompier, il sera même entouré par un deuxième cercle plus grand. Le premier cercle en bleu-gris, présent avec tous les agents, représente son champ de vision, tandis que le deuxième en mauve, présent uniquement chez les pompiers, représente son champ d'action pour l'extinction des bâtiments.

Les routes sont présentées de diverses manières en fonction de l'information qu'en a l'agent. Si la route n'a encore jamais été explorée, elle sera blanche et discontinue. Si elle a été aperçue et qu'elle est praticable, elle sera blanche et continue. Si par contre elle n'est pas utilisable, elle sera noire.

Les carrés représentant les bâtiments sont vides s'ils n'ont pas encore été aperçus ou qu'ils ne sont pas endommagés. Ils seront pleins dans le cas contraire. Un bâtiment en train d'être arrosé aura le contenu de son carré bleu, à condition d'avoir placé préalablement dans la méthode d'extinction le bout de code que voici : `if (USING_SCOPE && self().id() == SCOPING_ID) m_scope.set(SCOPE_FIRE_TARGET, target, Color.blue, -2) ;)`.

Lorsqu'un agent a déterminé le chemin le long duquel il compte se déplacer, ce chemin apparaît en rouge. Il restera en rouge, de son point initial à son point final, tant que l'agent n'aura pas déterminé un autre chemin, même si l'agent a fini son déplacement.

C.21.4 Zoom et décentrage

Un problème gênant qui a été relevé est le décentrage du *scope*. Par exemple, lorsqu'on agrandit la fenêtre horizontalement, seul le côté droit s'étend, et il n'est pas possible d'aller voir ce qui se passe plus à gauche. Voici quelques astuces pour résoudre ce problème :

Dans la classe Scope, une ligne ressemble à celle-ci :

```
static final int SIZE= 3 ;
```

Augmenter cette valeur provoque un agrandissement, et la diminuer, un rétrécissement. Nous l'avons réduit à 2. Les deux lignes suivantes permettent de centre l'image de la fenêtre :

```
private int x(int ox) { return compact( ox - 22910000 ) ; }
private int y(int oy) { return compact(122500 -
   (oy - 3667500)) ; }
```

Une diminution du nombre soustrait à `ox` crée un déplacement vers la gauche, tandis qu'une augmentation crée un déplacement vers la droite. De même, une diminution du nombre soustrait à `oy` crée un déplacement vers le haut, tandis qu'une augmentation crée un déplacement vers le bas. Notre fenêtre ne nous permettant pas de voir tout le côté gauche, nous avons diminué le nombre soustrait à `ox` à 22 740 000. Elle ne permettait pas non plus de voir le haut, aussi avons-nous augmenté le nombre soustrait à `oy` à 3 712 500.

Index

Annexe D

De YabAI en KADAI

Cette annexe présente la liste des différences entre le code KADAI réalisé lors du stage et le code YabAI (présenté à l'annexe C) :

1. **Predicate.java** Identique.
2. **Function.java** Identique.
3. **RescueObject.java** Bogue dans la méthode **isGap()** identifié et corrigé.
4. **Domain.java** Identique.
5. **ObjectPool.java** Bogue dans la méthode **lookAtOthers()** identifié et corrigé.
6. **CostFunction.java** Identique.
7. **Route.java** Identique.
8. **Router.java** Identique.
9. **IO.java** Adaptation de la méthode **tell()** à la classe Msg.
10. **Main.java** Identique.
11. **Message.java** Supprimée et remplacée par la classe **Msg.java** dont le but est identique.
12. **Action.java** Identique.
13. **History.java** Identique.
14. **Controller.java** Identique.
15. **Constants.java** Identique.
16. **CivilianController.java** Suprimée.
17. **KdTree.java** Suprimée.

18. **FireBrigadeController.java** Recréée entièrement.
19. **AmbulanceTeamController.java** Recréée entièrement.
20. **PoliceForceController.java** Recréée entièrement.
21. **FireStationController.java** Récréée entièrement.
22. **AmbulanceCenterController.java** Recréée entièrement.
23. **PoliceOfficeController.java** Recréée entièrement.
24. **Scope.java** Modifiée pour recentrer la carte et l'échelle et pour faire apparaître plus clairement les intentions de l'agent et les bâtiments en feu.

Annexe E

Résultats des simulations

Nous présentons dans cette annexe les résultats des simulations des agents YabAI et KADAI. Nous avons sélectionné cinq situations initiales (cartes) parmi celles qui ont été utilisées lors du championnat de 2001. Voici les cartes que nous avons utilisées, ainsi que leurs caractéristiques[1] :

	Nom	Nbr incendies	Nbr refuges	Commentaire
1.	comitee_map	4	7	Carte de base
2.	ISI_1	3	32	YabAI vainqueur
3.	JaistR_2	10	6	Incendies au SE ; policier au centre ; autres agents au NO
4.	JaistR_1	7	6	Incendies, pompiers et ambulances au NO ; policiers au centre
5.	gr2000_1	4	3	YabAI vainqueur

Voici les résultats que nous avons obtenus (V est l'évaluation totale, I est la surface incendiée en m^2, D est le nombre de dommages physiques et M le nombre de messages échangés). Nous n'avons pas commencé à mesurer de suite le nombre de messages échangés et nous avons réduit le nombre de tests après la première carte, considérant qu'une dizaine de tests étaient suffisants.

[1] Vous pouver les télécharger à l'adresse [Report, 2001]

comitee_map							
KADAI				YabAI			
V	I	D	M	V	I	D	M
22.4208769087	398615	2512		24.4874823103	423801	13378	
23.4312879262	408752	2547		25.5317313891	416781	26655	
24.4274487267	405709	2353		12.4652359510	237514	43469	
22.4268444715	404470	2520		25.5257971723	434014	21093	
19.4148484434	391826	2725		34.5303580643	464581	14696	
21.4108719021	389487	2324		36.6461757385	485264	43489	
19.4092881630	387145	2521		29.4829133123	431350	10252	
27.4621112376	437689	2954		28.4833384896	434018	9677	
17.3830397490	357236	3479		29.6625067189	426856	59174	
22.4348398480	404556	4541		30.5527411649	475076	18383	
20.4051821997	379789	3341	58	28.4773354693	427644	9714	2424
18.4021328769	379964	2551	11	30.5428391679	482553	13445	2241
21.4001064897	371200	4205	24	28.4932254182	418878	16095	798
22.4237812870	397525	3524	44	25.4472927732	407745	6922	2343
21.4026503501	380655	2506	33	25.4374093797	392691	8152	1846
22.4432158962	411290	4966	56	28.5111833371	444989	14429	1752
24.4150127362	390144	3190	116	30.5862277814	443020	35507	964
21.4006533172	378285	2605	29	28.5220299437	468200	11330	2346
25.4394852560	417651	2332	20	27.5212916896	446398	16856	1423
20.4094804525	387337	2520	116	30.4893032972	440967	9476	2504
21.4114999977	388187	2808	109	30.6258364050	447776	45432	1085
24.4122584300	385128	3759	38	28.5101990543	399999	24797	999
24.4239175494	403068	2136	99	21.4784414699	402156	16245	1982
21.4151675505	386194	4214	96	35.6198797626	445199	44300	1094
19.4436358717	410787	5204	31	40.6254101589	515064	29985	1003
20.4254011887	398852	3595	33	37.7375313622	497786	68644	1430
19.4535765195	425712	3901	120	26.4956294048	451426	8426	2804
22.4350856528	406972	3984	144	34.5587902256	421120	32811	1170
25.4606112270	426858	5467	25	35.6370195136	519940	32424	1011
24.4385568060	409529	4218	185	35.5168432454	460130	12038	2368

ISI_1							
KADAI				YabAI			
V	I	D	M	V	I	D	M
21.4543989690	440767	0	120	31.4692659793	455188	0	3216
26.4496649484	436175	0	88	30.4480639175	434622	0	3133
24.4394206185	426238	0	108	33.4980742268	483132	0	4401
25.4460000000	432620	0	48	29.4824649484	467991	0	3579
30.4925567010	477780	0	121	34.5158646153	497850	801	3262
24.4217742268	409121	0	33	38.5185144329	502959	0	3796
18.4091319587	396858	0	0	33.4835948453	469087	0	4231
19.4237371134	411025	0	127	34.5225333377	502666	1336	2306
22.4344628866	421429	0	0	34.4803463917	465936	0	2737
23.4137958762	401382	0	107	30.5063907216	491199	0	3431
JaistR_2							
KADAI				YabAI			
V	I	D	M	V	I	D	M
38.5995329702	522385	19687	79	47.6540342996	589803	17477	1617
37.5862074137	508269	19469	0	51.6839930319	608626	22607	1778
43.6486038486	548520	28493	75	56.9444239390	626181	20539	1874
37.6268168433	510682	31563	53	52.6931210377	605546	27293	1623
34.6210497188	511659	29495	20	47.6736794157	590807	24614	2185
43.6521308039	549321	29475	6	51.7687774546	614763	50908	2734
45.6490970437	561976	24695	251	54.6920432273	617447	22745	1962
42.5946783163	509930	21662	156	52.6940557734	622533	21734	2187
42.6164706723	554368	15628	136	53.6964553478	635878	17691	1862
46.6738053354	566324	32200	145	44.6547470998	575770	22398	1773
JaistR_1							
KADAI				YabAI			
V	I	D	M	V	I	D	M
35.5292816453	505049	2676	166	57.8099162889	632550	67565	2627
42.5595569623	529626	4446	124	46.7598605742	576476	60784	2762
38.5660522978	524217	8004	179	52.7895576026	625554	60678	2236
40.5744532533	534210	7864	195	51.7612095941	602405	55095	2209
37.5528358972	523257	4000	151	46.7004798014	606854	29783	1993
39.6538871433	522341	37243	192	47.7537678985	587419	55961	2340
42.5681885421	537583	4671	177	49.7723543972	609195	57792	1832
37.5425200861	512464	4415	183	48.7657933554	580662	62038	2009
45.7129828192	561853	47348	193	50.7613434428	602545	55112	2592
40.7405484217	550459	59601	23	50.7758707227	608294	59423	2553

gr2000_1							
KADAI				YabAI			
V	I	D	M	V	I	D	M
17.3710488493	239371	24575	23	23.5615189583	272389	58137	1607
17.5323608047	234827	57047	60	28.7422581401	307563	92736	1133
18.4270353617	240782	35428	24	27.6564455308	305746	74224	1691
21.4334811685	258564	33899	97	22.4905386263	277626	42638	1528
17.3190984288	222144	17404	43	23.6212934569	277901	69892	1456
16.5043444359	212194	54450	0	22.6151901407	268681	69674	1271
15.3020768424	218468	14775	21	25.6516267805	282081	75783	1137
				18.4093115981	264524	27977	954
				19.4559887809	252336	39429	1341
				19.5091202025	249059	50575	1062

Bibliographie

[Arc, 2001] (2001). RoboCupRescue Archive. http://www.robocup.org/games/362.html.

[BBC, 2001] BBC (2001). Robots aid New York rescue workers. *BBC News*. http://news.bbc.co.uk/hi/english/sci/tech/newsid_1548000/1548709.stm.

[Chaib-draa, 1999] Chaib-draa, B. (1999). *Agents et systèmes multiagents*. Département d'informatique, Faculté des sciences et de génie, Université Laval, Québec. Notes du cours IFT 64881A.

[Committee, 2000] Committee, T. R. R. T. (2000). *RoboCup-Rescue Simulator Manual Version 0 Revision 4*. http://robomec.cs.kobe-u.ac.jp/robocup-rescue.

[CRASAR, 2001] CRASAR (2001). Laboratoire de recherche CRASAR (Center for Robot-Assisted Search And Rescue). http://www.csee.usf.edu/robotics/crasar.

[Download, 2001] Download (2001). RoboCupRescue Simulation System Basic Package. http://ne.cs.uec.ac.jp/~koto/rescue.

[Georgeff and Lansky, 1987] Georgeff, M. and Lansky, A. (1987). Reactive reasoning and planning. In *The Proceedings of AAAI-87*, pages 677–682, Seattle.

[Jennings, 2000] Jennings, N. (2000). Agent methodology for software engineering. In *Communication of ACM*.

[Kahney, 2001] Kahney, L. (2001). Robots Scour WTC Wreckage. *Wired News*. http://www.wired.com/news/technology/0,1282,46930,00.html.

[Koch, 2002] Koch, E. (2002). Simulation multiagent de situations d'urgence. http://www.damas.ift.ulaval.ca/~koch.

[Lee, 2001] Lee, J. . (2001). Agile in a Crisis, Robots Show Their Mettle. *New York Times*. http://www.nytimes.com/2001/09/27/technology/circuits/27ROBO.html?todaysheadlines.

[Les Publications du Québec, 1999] Les Publications du Québec, editor (1999). *Pour affronter l'imprévisible. Les enseignements du verglas de 98. Rapport de la Commission scientifique et technique chargée d'analyser les évènements relatifs à la tempête de verglas survenue du 5 au 9 janvier 1998.* Les Publications du Québec, 1500 D, rue Jean-Talon Nord, Sainte-Foy (Québec) G1N 2E5.

[Proposal, 2001] Proposal (2001). RoboCup 2002 Rescue Simulation League Information. Proposal 01-12-2001. `http ://Kiyosu.isc.chubu.ac.jp/robocup/Rescue/2002memo/12_1_proposal.txt`.

[RCJ, 2001a] RCJ (2001a). RoboCup Junior 2001 Official web site. `http://www.demo.cs.brandeis.edu/rcj2001`.

[RCJ, 2001b] RCJ (2001b). RoboCup Junior Official Site. `http://www.robocupjunior.org`.

[RCR, 2001] RCR (2001). RoboCup-Rescue Official Web Page. `http://www.r.cs.kobe-u.ac.jp/robocup-rescue/`.

[RCR, 2002] RCR (2002). RoboCupRescue Robot League Home Page. `http://www.r.cs.kobe-u.ac.jp/robocup-rescue/rescue-robot.html`.

[RCRDamas, 2002] RCRDamas (2002). RobocupRescue au DAMAS. `http://www.damas.ift.ulaval.ca/projets/RobocupRescue/index.html`.

[RCS, 2001] RCS (2001). RoboCup Official Site. `http://www.robocup.org`.

[Report, 2001] Report (2001). RoboCup 2001 Seattle RoboCup-Rescue Simulation League Competition Report. `http://www.r.cs.kobe-u.ac.jp/robocup-rescue/robocup2001report.html`.

[Rules, 2002] Rules (2002). RoboCup 2002 Rescue Simulation Leagues. `http ://Kiyosu . isc . chubu . ac . jp/robocup/Rescue/2002memo/2002-rule-8.html`.

[SGI, 2000] SGI (2000). SGI to sponsor RoboCup. `http://www.hoise.com/primeur/00/articles/weekly/AE-PR-10-00-13.html`.

[Sobek, 1998] Sobek, D. R. P. (1998). ICRA (International Conference on Robotics and Automation) Home Page. `http://www.laas.fr/icra-98/icra-98.html`.

[Takeshi, 2001] Takeshi, M. (2001). Page des agents de l'équipe YabAI. `http://ne.cs.uec.ac.jp/~morimoto/rescue`.

[Takeshi et al., 2001] Takeshi, M., Kenji, K., and Ikuo, T. (2001). YabAI, The first Rescue Simulation League Champion. `http://www.r.cs.kobe-u.ac.jp/robocup-rescue/TPD01YabAI.pdf`.

[Trivedi, 2001] Trivedi, B. P. (2001). Search-and-Rescue Robots Tested at New York Disaster Site. *National Geographic*. `http://news.nationalgeographic.com/news/2001/09/0914_TVdisasterrobot.html`.

[Wooldridge et al., 1998] Wooldridge, M., Sycara, K., and Jennings, N. (1998). A roadmap of agent research and development. *Int Journal of Autonomous Agent and Multi-Agent Systems*, 1(1) :7–38.